

Hans Berger

M100.1

M BIT

Q

R

E32.1

POS

Automating with STEP 7 in LAD and FBD

SIMATIC S7-300/400
Programmable Controllers

SIEMENS

ZAEHLER

E3.4 — ZV

E5.2 — ZR

M2.0 — MP

D — W30

DB10.DBW12 — IN IN1

C#123 — W

DB10.DBW20 — IN IN2

E10.0 — SUB

DB10.DBW20 — IN IN1

OUT — #t_INT

DB10.DBW14 — IN IN2

DB10.DBW12 — IN IN2

E5.0

A7.3

E1.3

E1.4

E1.6

E1.7

E1.5

A4.2

()

>=1

A3.0

M3.1

&

Fifth Edition

Berger Automating with STEP 7 in LAD and FBD

Automating with STEP 7 in LAD and FBD

SIMATIC S7-300/400
Programmable Controllers

by Hans Berger

5th revised and enlarged edition, 2012

Publicis Publishing

Bibliographic information published by the Deutsche Nationalbibliothek

The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie;
detailed bibliographic data are available in the Internet at <http://dnb.d-nb.de>.

This book contains one Trial DVD **“SIMATIC STEP 7 Professional, Edition 2010 SR1, Trial License”**
encompasses: SIMATIC STEP 7 V5.5 SP1, S7-GRAPH V5.3 SP7, S7-SCL V5.3 SP6, S7-PLCSIM V5.4 SP5
and can be used for trial purposes for 14 days.

This Software can only be used with the Microsoft Windows XP 32 Bit Professional Edition SP3 or
Microsoft Windows 7 32/64 Bit Professional Edition SP1 or Microsoft Windows 7 32/64 Bit Ultimate
Edition SP1 operating systems.

Additional information can be found in the internet at:

www.siemens.com/sce/contact

www.siemens.com/sce/modules

www.siemens.com/sce/tp

The programming examples concentrate on describing the LAD and FBD functions and providing
SIMATIC S7 users with programming tips for solving specific tasks with this controller.

The programming examples given in the book do not pretend to be complete solutions or to be executable
on future STEP 7 releases or S7-300/400 versions. Additional care must be taken in order to comply with
the relevant safety regulations.

The author and publisher have taken great care with all texts and illustrations in this book. Nevertheless,
errors can never be completely avoided. The publisher and the author accept no liability, regardless of
legal basis, for any damage resulting from the use of the programming examples.

The author and publisher are always grateful to hear your responses to the contents of the book.

Publicis Publishing

P.O. Box 3240

91050 Erlangen

E-mail: publishing-distribution@publicis.de

Internet: www.publicis-books.de

ISBN 978-3-89578-410-1

5th revised and enlarged edition, 2012

Editor: Siemens Aktiengesellschaft, Berlin and Munich

Publisher: Publicis Publishing, Erlangen

© 2012 by Publicis Erlangen, Zweigniederlassung der PWW GmbH

This publication and all parts thereof are protected by copyright. Any use of it outside the
strict provisions of the copyright law without the consent of the publisher is forbidden and will
incur penalties. This applies particularly to reproduction, translation, microfilming or other
processing, and to storage or processing in electronic systems. It also applies to the use of
individual illustrations or extracts from the text.

Printed in Germany

Preface

The SIMATIC automation system unites all the subsystems of an automation solution under uniform system architecture into a homogeneous whole from the field level right up to process control. This Totally Integrated Automation (TIA) concept permits integrated configuring, programming, data management and communications within the complete automation system. Fine-tuned communications mechanisms permit harmonious interaction between programmable controllers, visualization systems and distributed I/Os.

As the basic tool for SIMATIC, STEP 7 handles the parenthesis function for Totally Integrated Automation. STEP 7 is used to carry out the configuration and programming of the SIMATIC S7, SIMATIC C7 and SIMATIC WinAC automation systems. Microsoft Windows has been selected as the operating system, thus opening up the world of standard PCs with the user desktop widely used in the office environment.

For block programming STEP 7 provides programming languages that comply with DIN EN 6.1131-3: STL (statement list; an Assembler-like language), LAD (ladder logic; a representation similar to relay logic diagrams), FBD (function block diagram) and the S7-SCL optional package (structured control language, a Pascal-like high-level language). Several optional packages supplement these languages: S7-GGRAPH (sequential control), S7-HiGraph (programming with state-transition diagrams) and CFC (connecting blocks; similar to function block diagram). The various methods of representation allow every user to select the suitable control function description. This

broad adaptability in representing the control task to be solved significantly simplifies working with STEP 7.

This book describes the LAD and FBD programming languages for S7-300/400. As a valuable supplement to the language description, and following an introduction to the S7-300/400 automation system, it provides valuable and practice-oriented information on the basic handling of STEP 7 for the configuration of SIMATIC PLCs, their networking and programming. The description of the “basic functions” of a binary control, such as e.g. logic operations or storage functions, is particularly useful for beginners or those converting from contactor controls to STEP 7. The digital functions explain how digital values are combined; for example, basic calculations, comparisons or data type conversion.

The book shows how you can control the program processing (program flow) with LAD and FBD and design structured programs. In addition to the cyclically processed main program, you can also incorporate event-driven program sections as well as influence the behavior of the controller at startup and in the event of errors/faults. The book concludes with a general overview of the system functions and the function set for LAD and FBD. The contents of this book describe Version 5.5 of the STEP 7 programming software.

Erlangen, January 2012

Hans Berger

The Contents of the Book at a Glance

Overview of the S7-300/400 programmable logic controller

PLC functions comparable to a contactor control system

Handling numbers and digital operands

Introduction	Basic functions	Digital functions
1 SIMATIC S7-300/400 Programmable Controller Structure of the Programmable Controller (Hardware Components of S7-300/400); Memory Areas; Distributed I/O (PROFIBUS DP); Communications (Subnets); Module Addresses; Addresses Areas	4 Binary Logic Operations AND, OR and Exclusive OR Functions; Nesting Functions	9 Comparison Functions Comparison According to Data Types INT, DINT and REAL
2 STEP 7 Programming Software Editing Projects; Configuring Stations; Configuring the Network; Symbol Editor; LAD/FBD Program Editor; Online Mode; Testing LAD and FBD Programs	5 Memory Functions Assign, Set and Reset; Midline Outputs; Edge Evaluation; Example of a Conveyor Belt Control System	10 Arithmetic Functions Four-function Math with INT, DINT and REAL numbers;
3 SIMATIC S7 Program Program Processing; Block Types; Programming Code Blocks and Data Blocks; Addressing Variables, Constant Representation, Data Types Description	6 Move Functions Load and Transfer Functions; System Functions for Data Transfer	11 Mathematical Functions Trigonometric Functions; Arc Functions; Squaring, Square-root Extraction, Exponentiation, Logarithms
	7 Timers Start SIMATIC Timers with Five Different Characteristics, Resetting and Scanning; IEC Timer Functions	12 Conversion Functions Data Type Conversion; Complement Formation
	8 Counters SIMATIC Counters; Count up, Count down, Set, Reset and Scan Counters; IEC Counter Functions	13 Shift Functions Shifting and Rotating
		14 Word Logic Processing a AND, OR and Exclusive OR Word Logic Operation

Controlling
program execution,
block functions

Processing
the user program

Supplements to LAD and
FBD; block libraries,
Function overviews

Program Flow Control

15 Status Bits

Binary Flags,
Digital Flags;
Setting and Evaluating the
Status Bits;
EN/ENO Mechanism

16 Jump Functions

Unconditional Jump;
Jump if RLO = "1"
Jump if RLO = "0"

17 Master Control Relay

MCR Dependency,
MCR Area,
MCR Zone

18 Block Functions

Block Call,
Block End;
Temporary and Static Local
Data, Local Instances;
Accessing Data Operands
Opening a Data Block

19 Block Parameters

Formal Parameters,
Actual Parameters;
Declarations and Assignments,
"Parameter Passing"

Program Processing

20 Main Program

Program Structure;
Scan Cycle Control
(Response Time,
Start Information,
Background Scanning);
Program Functions;
Communications with
PROFIBUS and PROFINET;
GD Communications;
S7 and S7 Basic
Communications

21 Interrupt Handling

Hardware Interrupts;
Watchdog Interrupts;
Time-of-Day Interrupts;
Time-Delay Interrupts;
DPV1 Interrupts
Multiprocessor Interrupt;
Handling Interrupt Events

22 Restart Characteristics

Cold Restart, Warm Restart,
Hot Restart;
STOP, HOLD, Memory Reset;
Parameterizing Modules

23 Error Handling

Synchronous Errors;
Asynchronous Errors;
System Diagnostics

Appendix

24 Supplements to Graphic Programming

Block Protection
KNOW_HOW_PROTECT;
Indirect Addressing,
Pointers: General Remarks;
Brief Description of the
"Message Frame Example"

25 Block Libraries

Organization Blocks;
System Function Blocks;
IEC Function Blocks;
S5-S7 Converting Blocks;
TI/S7 Converting Blocks;
PID Control Blocks;
Communication Blocks

26 Function Set LAD

Basic Functions;
Digital Functions;
Program Flow Control

27 Function Set FBD

Basic Functions;
Digital Functions;
Program Flow Control

The Programming Examples at a Glance

The present book provides many figures representing the use of the LAD and FBD programming languages. All programming examples can be downloaded from the publisher's website www.publicis.de/books. There are two libraries LAD_Book and FBD_Book.

The libraries LAD_Book and FBD_Book contain eight programs that are essentially illustrations of the graphical representation. Two extensive examples show the programming of functions, function blocks and local instances (Conveyor Example) and the handling of data (Message Frame Example). All the examples contain symbols and comments.

Library LAD_Book

Data Types Examples of Definition and Application	Program Processing Examples of SFC Calls
FB 101 Elementary Data Types FB 102 Complex Data Types FB 103 Parameter Types	FB 120 Chapter 20: Main Program FB 121 Chapter 21: Interrupt Processing FB 122 Chapter 22: Start-up Characteristics FB 123 Chapter 23: Error Handling
Basic Functions LAD Representation Examples	Conveyor Example Examples of Basic Functions and Local Instances
FB 104 Chapter4: Series and Parallel Circuits FB 105 Chapter5: Memory Functions FB 106 Chapter6: Move Functions FB 107 Chapter7: Timer Functions FB 108 Chapter8: Counter Functions	FC 11 Belt Control FC 12 Counter Control FB 20 Feed FB 21 Conveyor Belt FB 22 Parts Counter
Digital Functions LAD Representation Examples	Message Frame Example Data Handling Examples
FB 109 Chapter 9: Comparison Functions FB 110 Chapter 10: Arithmetic Functions FB 111 Chapter 11: Math Functions FB 112 Chapter 12: Conversion Functions FB 113 Chapter 13: Shift Functions FB 114 Chapter 14: Word Logic	UDT 51 Data Structure for the Frame Header UDT 52 Data Structure for a Message FB 51 Generate Message Frame FB 52 Store Message Frame FC 51 Time-of-day Check FC 52 Copy Data Area with indirect Addressing
Program Flow Control LAD Representation Examples	General Examples
FB 115 Chapter 15: Status Bits FB 116 Chapter 16: Jump Functions FB 117 Chapter 17: Master Control Relay FB 118 Chapter 18: Block Functions FB 119 Chapter 19: Block Parameters	FC 41 Range Monitor FC 42 Limit Value Detection FC 43 Compound Interest Calculation FC 44 Doubleword-wise Edge Evaluation

The libraries are supplied in archived form. Before you can start working with them, you must dearchive the libraries. Select the FILE → DEARCHIVE menu item in the SIMATIC Manager and follow the instructions (see also the README.TXT within the download files).

To try the programs out, set up a project corresponding to your hardware configuration and then copy the program, including the symbol table from the library to the project. Now you can call the example programs, adapt them for your own purposes and test them online.

Library FBD_Book

Data Types Examples of Definition and Application	Program Processing Examples of SFC Calls
FB 101 Elementary Data Types FB 102 Complex Data Types FB 103 Parameter Types	FB 120 Chapter 20: Main Program FB 121 Chapter 21: Interrupt Processing FB 122 Chapter 22: Start-up Characteristics FB 123 Chapter 23: Error Handling
Basic Functions FBD Representation Examples	Conveyor Example Examples of Basic Functions and Local Instances
FB 104 Chapter4: Series and Parallel Circuits FB 105 Chapter5: Memory Functions FB 106 Chapter6: Move Functions FB 107 Chapter7: Timer Functions FB 108 Chapter8: Counter Functions	FC 11 Belt Control FC 12 Counter Control FB 20 Feed FB 21 Conveyor Belt FB 22 Parts Counter
Digital Functions FBD Representation Examples	Message Frame Example Data Handling Examples
FB 109 Chapter 9: Comparison Functions FB 110 Chapter 10: Arithmetic Functions FB 111 Chapter 11: Math Functions FB 112 Chapter 12: Conversion Functions FB 113 Chapter 13: Shift Functions FB 114 Chapter 14: Word Logic	UDT 51 Data Structure for the Frame Header UDT 52 Data Structure for a Message FB 51 Generate Message Frame FB 52 Store Message Frame FC 51 Time-of-day Check FC 52 Copy Data Area with indirect Addressing
Program Flow Control FBD Representation Examples	General Examples
FB 115 Chapter 15: Status Bits FB 116 Chapter 16: Jump Functions FB 117 Chapter 17: Master Control Relay FB 118 Chapter 18: Block Functions FB 119 Chapter 19: Block Parameters	FC 41 Range Monitor FC 42 Limit Value Detection FC 43 Compound Interest Calculation FC 44 Doubleword-wise Edge Evaluation

Automating with STEP 7

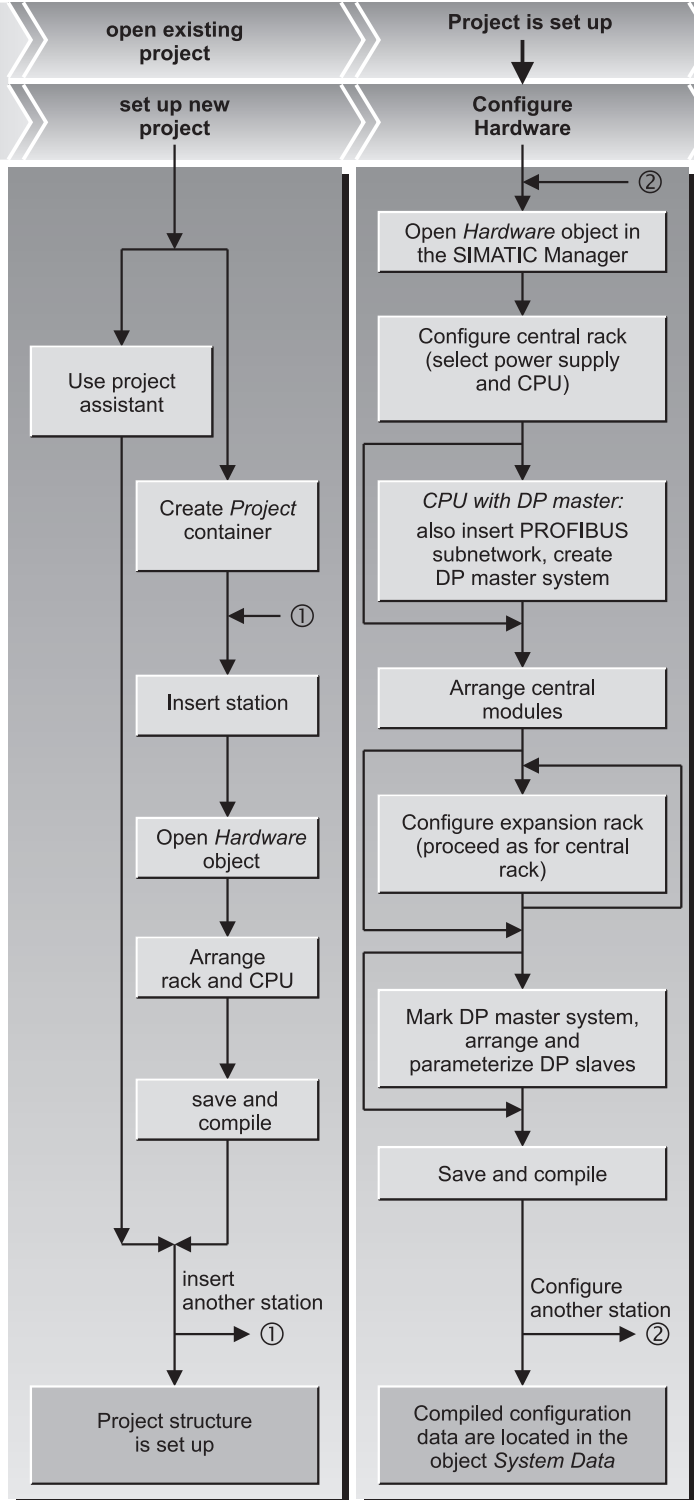
This double page shows the basic procedure for using the STEP 7 programming software.

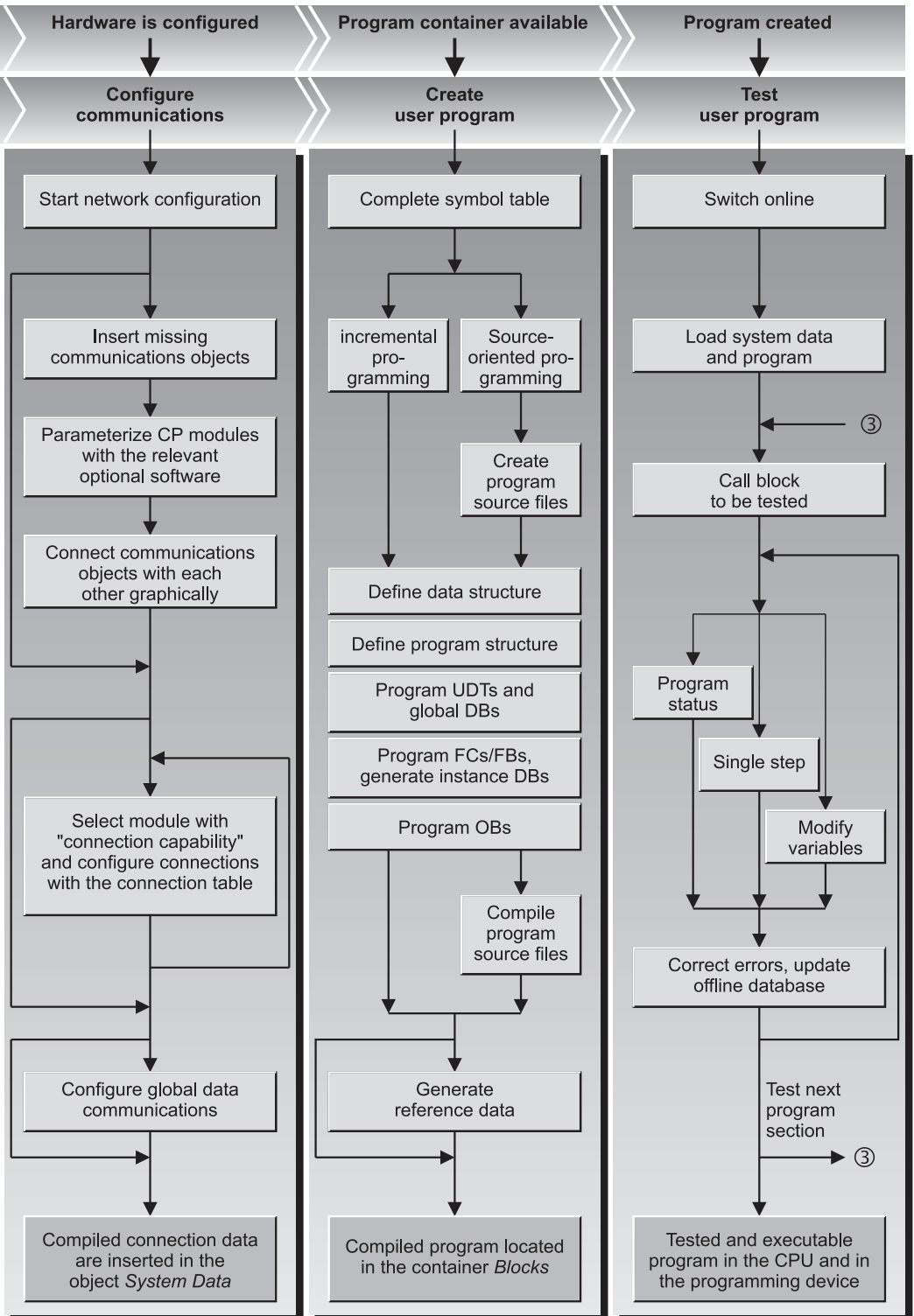
Start the SIMATIC Manager and set up a new project or open an existing project. All the data for an automation task are stored in the form of *objects* in a *project*. When you set up a project, you create *containers* for the accumulated data by setting up the required *stations* with at least the CPUs; then the containers for the user programs are also created. You can also create a *program container* direct in the project.

In the next steps, you configure the hardware and, if applicable, the communications connections. Following this, you create and test the program.

The order for creating the automation data is not fixed. Only the following general regulation applies: if you want to process objects (data), they must exist; if you want to insert objects, the relevant containers must be available.

You can interrupt processing in a project at any time and continue again from any location the next time you start the SIMATIC Manager.





Contents

Introduction	19	2	STEP 7 Programming Software	49
1	SIMATIC S7-300/400			
	Programmable Controller			20
1.1	Structure of the Programmable Controller		2.1	STEP 7 Basis Package 49
			2.1.1	Installation 49
1.1.1	Components		2.1.2	Automation License Manager 50
1.1.2	S7-300 Station		2.1.3	SIMATIC Manager 50
1.1.3	S7-400 station		2.1.4	Projects and Libraries 53
1.1.4	Fault-tolerant SIMATIC		2.1.5	Multiprojects 54
1.1.5	Safety-related SIMATIC		2.1.6	Online Help 54
1.1.6	CPU Memory Areas		2.2	Editing Projects 54
1.2	Distributed I/O		2.2.1	Creating Projects 54
			2.2.2	Managing, Reorganizing and Archiving 56
1.2.1	PROFIBUS DP		2.2.3	Project Versions 57
1.2.2	PROFINET IO		2.2.4	Creating and editing multiprojects 57
1.2.3	Actuator/Sensor Interface		2.3	Configuring Stations 58
1.2.4	Gateways		2.3.1	Arranging Modules 60
1.3	Communications		2.3.2	Addressing Modules 60
			2.3.3	Parameterizing Modules 61
1.3.1	Introduction		2.3.4	Networking Modules with MPI 61
1.3.2	Subnets		2.3.5	Monitoring and Modifying Modules 62
1.3.3	Communications Services		2.4	Configuring the Network 62
1.3.4	Connections		2.4.1	Configuring the Network View 64
1.4	Module Addresses		2.4.2	Configuring a Distributed I/O with the Network Configuration 64
			2.4.3	Configuring connections 65
1.4.1	Signal Path		2.4.4	Gateways 68
1.4.2	Slot Address		2.4.5	Loading the Connection Data 69
1.4.3	Logical Address		2.4.6	Matching Projects in a Multiproject 69
1.4.4	Module Start Address		2.5	Creating the S7 Program 71
1.4.5	Diagnostics Address			
1.4.6	Addresses for Bus Nodes		2.5.1	Introduction 71
1.5	Address Areas		2.5.2	Symbol Table 71
			2.5.3	Program Editor 73
1.5.1	User Data Area		2.5.4	Rewiring 77
1.5.2	Process Image		2.5.5	Address Priority 77
1.5.3	Consistent User Data		2.5.6	Reference Data 78
1.5.4	Bit Memories		2.5.7	Language Setting 80

12.2	Conversion of INT and DINT Numbers	209	18	Block Functions	235
12.3	Conversion of BCD Numbers . .	210	18.1	Block Functions for Code Blocks.	235
12.4	Conversion of REAL Numbers .	210	18.1.1	Block Calls: General	236
12.5	Miscellaneous Conversion Functions.	212	18.1.2	Call Box.	237
13	Shift Functions	213	18.1.3	CALL Coil/Box.	238
13.1	Processing a Shift Function . . .	213	18.1.4	Block End Function.	239
13.2	Shift	215	18.1.5	Temporary Local Data	240
13.3	Rotate	216	18.1.6	Static Local Data	241
14	Word Logic	217	18.2	Block Functions for Data Blocks .	244
14.1	Processing a Word Logic Operation.	217	18.2.1	Two Data Block Registers	244
14.2	Description of the Word Logic Operations	219	18.2.2	Accessing Data Operands.	245
			18.2.3	Opening a Data Block.	246
			18.2.4	Special Points in Data Addressing	247
			18.3	System Functions for Data Blocks	248
			18.3.1	Creating a Data Block in Work Memory.	249
			18.3.2	Creating a Data Block in Load Memory.	250
			18.3.3	Deleting a Data Block.	251
			18.3.4	Testing a Data Block	251
	Program Flow Control	220	19	Block Parameters	252
15	Status Bits	221	19.1	Block Parameters in General . . .	252
15.1	Description of the Status Bits . .	221	19.1.1	Defining the Block Parameters . .	252
15.2	Setting the Status Bits	222	19.1.2	Processing the Block Parameters .	253
15.3	Evaluating the Status Bits	224	19.1.3	Declaration of the Block Parameters	253
15.4	Using the Binary Result	225	19.1.4	Declaration of the Function Value	254
15.4.1	Setting the Binary Result BR . .	225	19.1.5	Initializing Block Parameters . . .	254
15.4.2	Main Rung, EN/ENO Mechanism	225	19.2	Formal Parameters	255
15.4.3	ENO in the Case of User-written Blocks	226	19.3	Actual Parameters.	257
16	Jump Functions.	227	19.4	“Forwarding” Block Parameters .	260
16.1	Processing a Jump Function . . .	227	19.5	Examples	260
16.2	Unconditional Jump	228	19.5.1	Conveyor Belt Example.	260
16.3	Jump if RLO = “1”.	229	19.5.2	Parts Counter Example	261
16.4	Jump if RLO = “0”.	229	19.5.3	Feed Example.	262
17	Master Control Relay	230		Program Processing.	269
17.1	MCR Dependency	230	20	Main Program	270
17.2	MCR Area	231	20.1	Program Organization.	270
17.3	MCR Zone	232	20.1.1	Program Structure.	270
17.4	Setting and Resetting I/O Bits . .	233	20.1.2	Program Organization.	271

20.2	Scan Cycle Control	272	20.7	S7 Communication	341
20.2.1	Process Image Updating.	272	20.7.1	Fundamentals.	341
20.2.2	Scan Cycle Monitoring Time	274	20.7.2	Two-Way Data Exchange.	342
20.2.3	Minimum Scan Cycle Time, Background Scanning	275	20.7.3	One-Way Data Exchange.	344
20.2.4	Response Time	276	20.7.4	Transferring Print Data	345
20.2.5	Start Information	276	20.7.5	Control Functions.	346
20.3	Program Functions	278	20.7.6	Monitoring Functions.	346
20.3.1	Time of day	278	20.8	IE communication	350
20.3.2	Read System Clock	280	20.8.1	Basics.	350
20.3.3	Run-Time Meter	280	20.8.2	Establishing and clearing down connections.	351
20.3.4	Compressing CPU Memory	282	20.8.3	Data transfer with TCP native or ISO-on-TCP	353
20.3.5	Waiting and Stopping	282	20.8.4	Data transfer with UDP.	355
20.3.6	Multicomputing	282	20.9	PtP communication with S7-300C	357
20.3.7	Determining the OB Program Runtime	283	20.9.1	Fundamentals.	357
20.3.8	Changing program protection	286	20.9.2	ASCII driver and 3964(R) procedure	358
20.4	Communication via Distributed I/O	287	20.9.3	RK512 computer coupling	359
20.4.1	Addressing PROFIBUS DP	287	20.10	Configuration in RUN	362
20.4.2	Configuring PROFIBUS DP	291	20.10.1	Preparation of Changes in Configuration	362
20.4.3	Special Functions for PROFIBUS DP	300	20.10.2	Change Configuration	364
20.4.4	Addressing PROFINET IO	305	20.10.3	Load Configuration	364
20.4.5	Configuring PROFINET IO	307	20.10.4	CiR Synchronization Time	365
20.4.6	Special Functions for PROFINET IO.	314	20.10.5	Effects on Program Execution.	365
20.4.7	System blocks for distributed I/O	323	20.10.6	Control CiR Process.	365
20.5	Global Data Communication	331	21	Interrupt Handling	367
20.5.1	Fundamentals	331	21.1	General Remarks	367
20.5.2	Configuring GD communication	333	21.2	Time-of-Day Interrupts.	368
20.5.3	System Functions for GD Communication	335	21.2.1	Handling Time-of-Day Interrupts	369
20.6	S7 Basic Communication	335	21.2.2	Configuring Time-of-Day Interrupts with STEP 7	370
20.6.1	Station-Internal S7 Basic Communication	335	21.2.3	System Functions for Time-of-Day Interrupts	370
20.6.2	System Functions for Station- Internal S7 Basic Communication	336	21.3	Time-Delay Interrupts	372
20.6.3	Station-External S7 Basic Communication	338	21.3.1	Handling Time-Delay Interrupts	372
20.6.4	System Functions for Station- External S7 Basic Communication	339	21.3.2	Configuring Time-Delay Interrupts with STEP 7	373
			21.3.3	System Functions for Time-Delay Interrupts	373

21.4	Watchdog Interrupts	374	22.5	Parameterizing Modules	397
21.4.1	Handling Watchdog Interrupts . .	375	22.5.1	General remarks on parameterizing modules.	397
21.4.2	Configuring Watchdog Interrupts with STEP 7	376	22.5.2	System Blocks for Module Parameterization	399
21.5	Hardware Interrupts	376	22.5.3	Blocks for Transmitting Data Records	401
21.5.1	Generating a Hardware Interrupt	376	23	Error Handling	404
21.5.2	Servicing Hardware Interrupts . .	377	23.1	Synchronous Errors	404
21.5.3	Configuring Hardware Interrupts with STEP 7	378	23.2	Synchronous Error Handling . .	406
21.6	DPV1 Interrupts	378	23.2.1	Error Filters	406
21.7	Multiprocessor Interrupt	380	23.2.2	Masking Synchronous Errors . .	407
21.8	Synchronous Cycle Interrupts . .	381	23.2.3	Unmasking Synchronous Errors .	408
21.8.1	Processing the Synchronous Cycle Interrupts	381	23.2.4	Reading the Error Register	408
21.8.2	Isochrone Updating Of Process Image.	382	23.2.5	Entering a Substitute Value	408
21.8.3	Configuration of Synchronous Cycle Interrupts with STEP 7 . .	383	23.3	Asynchronous Errors	408
21.9	Handling Interrupt Events	383	23.4	System Diagnostics	411
21.9.1	Disabling and Enabling interrupts	383	23.4.1	Diagnostic Events and Diagnostic Buffer	411
21.9.2	Delaying and Enabling Interrupts	384	23.4.2	Writing User Entries in the Diagnostic Buffer	411
21.9.3	Reading additional Interrupt Information.	385	23.4.3	Evaluating Diagnostic Interrupts .	412
22	Start-up Characteristics	387	23.4.4	Reading the System Status List . .	412
22.1	General Remarks.	387	23.5	Web Server	415
22.1.1	Operating Modes.	387	23.5.1	Activating the Web server	415
22.1.2	HOLD Mode.	388	23.5.2	Reading out Web information. . .	415
22.1.3	Disabling the Output Modules . .	388	23.5.3	Web information	415
22.1.4	Restart Organization Blocks . . .	388	Appendix	417	
22.2	Power-Up	389	24	Supplements to Graphic Programming	418
22.2.1	STOP Mode	389	24.1	Block Protection	418
22.2.2	Memory Reset	389	24.2	Indirect Addressing	419
22.2.3	Restoring the factory settings . .	390	24.2.1	Pointers: General Remarks	419
22.2.4	Retentivity	390	24.2.2	Area Pointer.	419
22.2.5	Restart Parameterization.	390	24.2.3	DB Pointer	419
22.3	Types of Restart	391	24.2.4	ANY Pointer	421
22.3.1	START-UP Mode	391	24.2.5	“Variable” ANY Pointer	421
22.3.2	Cold Restart	393	24.3	Brief Description of the “Message Frame Example”	422
22.3.3	Warm Restart.	393			
22.3.4	Hot Restart	394			
22.4	Ascertaining a Module Address .	394			

25	Block Libraries	426		
25.1	Organization Blocks	426		
25.2	System Function Blocks	427		
25.3	IEC Function Blocks	430		
25.4	S5-S7 Converting Blocks	431		
25.5	TI-S7 Converting Blocks	432		
25.6	PID Control Blocks	433		
25.7	Communication Blocks	433		
25.8	Miscellaneous Blocks	434		
25.9	SIMATIC_NET_CP	434		
25.10	Redundant IO MGP V31	435		
25.11	Redundant IO CGP V40	435		
25.12	Redundant IO CGP V51	435		
26	Function Set LAD	436		
26.1	Basic Functions	436		
26.2	Digital Functions	437		
26.3	Program Flow Control	439		
27	Function Set FBD	440		
27.1	Basic Functions	440		
27.2	Digital Functions	441		
27.3	Program Flow Control	443		
	Index444		
	Abbreviations451		

Introduction

This portion of the book provides an overview of the SIMATIC S7-300/400.

The **S7-300/400 programmable controller** is of modular design. The modules with which it is configured can be central (in the vicinity of the CPU) or distributed without any special settings or parameter assignments having to be made. In SIMATIC S7 systems, distributed I/O is an integral part of the system. The CPU, with its various memory areas, forms the hardware basis for processing of the user programs. A load memory contains the complete user program: the parts of the program relevant to its execution at any given time are in a work memory whose short access times are the prerequisite for fast program processing.

STEP 7 is the programming software for S7-300/400 and the automation tool is the SIMATIC Manager. The SIMATIC Manager is an application for the Windows operating systems from Microsoft and contains all functions needed to set up a project. When necessary, the SIMATIC Manager starts additional tools, for example to configure stations, initialize modules, and to write and test programs.

You formulate your automation solution in the STEP 7 programming languages. The **SIMATIC S7 program** is structured, that is to say, it consists of blocks with defined functions that are composed of networks or rungs. Different priority classes allow a graduated interruptibility of the user program currently executing. STEP 7 works with variables of various data types starting with binary variables (data type BOOL) through digital variables (e.g. data type INT or REAL for computing tasks) up to complex data types such as arrays or structures (combinations of variables of different types to form a single variable).

The first chapter contains an overview of the hardware in an S7-300/400 programmable controller, and the second chapter contains an overview of the STEP 7 programming software. The basis for the description is the function scope for STEP 7 Version 5.5

Chapter 3 “SIMATIC S7 Program” serves as an introduction to the most important elements of an S7 program and shows the programming of individual blocks in the programming languages LAD and FBD. The functions and operations of LAD and FBD are then described in the subsequent chapters of the book. All the descriptions are explained using brief examples.

- 1 **SIMATIC S7-300/400 Programmable Controller**
Structure of the programmable controller; distributed I/O; communications; module addresses; operand areas
- 2 **STEP 7 Programming Software**
SIMATIC Manager; processing a project; configuring a station; configuring a network; writing programs (symbol table, program editor); switching online; testing programs
- 3 **SIMATIC S7 Program**
Program processing with priority classes; program blocks; addressing variables; programming blocks with LAD and FBD; variables and constants; data types (overview)

1 SIMATIC S7-300/400 Programmable Controller

1.1 Structure of the Programmable Controller

1.1.1 Components

The SIMATIC S7-300/400 is a modular programmable controller comprising the following components:

- ▷ Racks
Accommodate the modules and connect them to each other
- ▷ Power supply (PS);
Provides the internal supply voltages
- ▷ Central processing unit (CPU)
Stores and processes the user program
- ▷ Interface modules (IMs);
Connect the racks to one another
- ▷ Signal modules (SMs);
Adapt the signals from the system to the internal signal level or control actuators via digital and analog signals
- ▷ Function modules (FMs);
Execute complex or time-critical processes independently of the CPU
- ▷ Communications processors (CPs)
Establish the connection to subsidiary networks (subnets)
- ▷ Subnets
Connect programmable controllers to each other or to other devices

A programmable controller (or station) may consist of several racks, which are linked to one another via bus cables. The power supply, CPU and I/O modules (SMs, FMs and CPs) are plugged into the central rack. If there is not enough room in the central rack for the I/O modules or if you want some or all I/O modules to be separate from the central rack, expansion racks are available which are connected to the central rack via interface modules (Figure 1.1).

It is also possible to connect distributed I/O to a station (see Chapter 1.2.1 “PROFIBUS DP”).

The racks connect the modules with two buses: the I/O bus (or P bus) and the communication bus (or K bus). The I/O bus is designed for high-speed exchange of input and output signals, the communication bus for the exchange of large amounts of data. The communication bus connects the CPU and the programming device interface (MPI) with function modules and communications processors.

1.1.2 S7-300 Station

Centralized configuration

In an S7-300 controller, as many as 8 I/O modules can be plugged into the central rack. Should this single-tier configuration prove insufficient, you have two options for controllers equipped with a CPU 313 or higher:

- ▷ A two-tier configuration (with IM 365 up to 1 meter between racks) or
- ▷ A configuration of up to four tiers (with IM 360 and IM 361 up to 10 meters between racks)

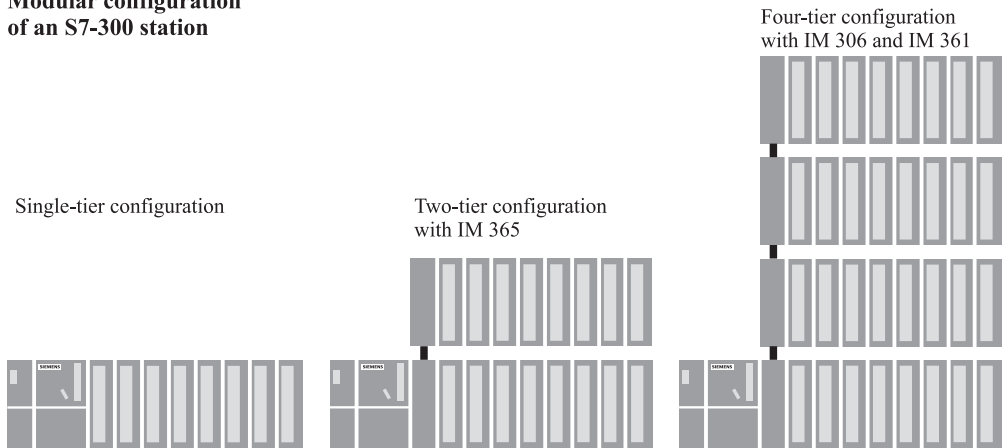
You can operate a maximum of 8 modules in a rack. The number of modules may be limited by the maximum permissible current per rack, which is 1.2 A.

The modules are linked to one another via a backplane bus, which combines the functions of the P and K buses.

Local bus segment

A special feature regarding configuration is the use of the FM 356 application module. An FM 356 is able to “split” a module’s backplane bus and to take over control of the remaining modules in the split-off “local bus segment” itself. The limitations mentioned above regarding the

Modular configuration of an S7-300 station



Modular configuration of an S7-400 station

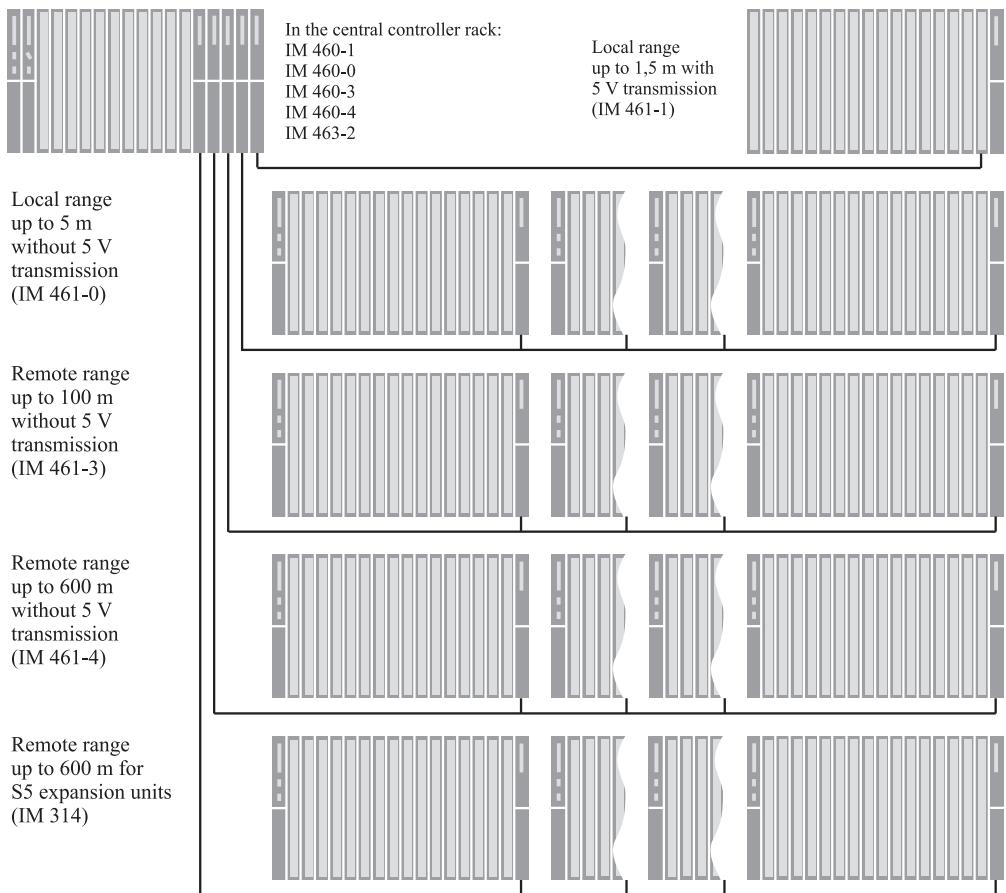


Figure 1.1 Hardware Configuration for S7-300/400

number of modules and the power consumption also apply in this case.

Standard CPUs

The standard CPUs are available in versions that differ with regard to memory capacity and processing speed. They range from the “smallest” CPU 312 for lower-end applications with moderate processing speed requirements, up to the CPU 319-3 PN/DP with its large program memory and high processing performance for cross-sector automation tasks. Equipped with the relevant interfaces, some CPUs can be used for central control of the distributed I/O via PROFIBUS and PROFINET. A micro memory card (MMC) is required for operating the standard CPUs – as with all innovated S7-300-CPU. This medium opens up new application possibilities compared to the previously used memory card (see Chapter 1.1.6 “CPU Memory Areas”).

The now discontinued CPU 318 can be replaced by the CPUs 317 and 319.

Compact CPUs

The 3xxC CPUs permit construction of compact mini programmable controllers. Depending on the version, they already contain:

- ▷ Integral I/Os
Digital and analog inputs/outputs
- ▷ Integral technology functions
Counting, measurement, control, positioning
- ▷ Integral communications interfaces
PROFIBUS DP master or slave, point-to-point coupling (PtP)

The technological functions are system blocks which use the onboard I/O of the CPU.

Technology CPUs

The CPUs 3xxT combine open-loop control functions with simple motion control functions. The open-loop control component is designed as in a standard CPU. It is configured, parameterized and programmed using STEP 7. The technology objects and the motion control component require the optional S7-Technology

package that is integrated in the SIMATIC Manager after installation.

The Technology CPUs have a PROFIBUS DP interface that allows operation as DP master or DP slave. The CPUs are used for cross-sector automation tasks in series mechanical equipment manufacture, special mechanical equipment manufacture, and plant building.

Failsafe CPUs

The CPUs 3xxF are used in production plants with increased safety requirements. The relevant PROFIBUS and PROFINET interfaces allow the operation of safety-related distributed I/O using the PROFIsafe bus profile (see “Safety Integrated for the manufacturing industry” under 1.1.5 “Safety-related SIMATIC”). Standard modules for normal applications can be used parallel to safety-related operation..

SIPLUS

The SIPLUS product family offers modules that can be used in harsh environments. The SIPLUS components are based on standard devices which have been specially converted for the respective application, for example for an extended temperature range, increased resistance to vibration and shock, or voltage ranges differing from the standard. Please therefore note the technical data for the respective SIPLUS module. In order to carry out the configuration with STEP 7, use the equivalent type (the standard module on which it is based); this is specified, for example, on the module's nameplate.

1.1.3 S7-400 station

Centralized configuration

The controller rack for the S7-400 is available in the UR1 (18 slots), UR2 (9 slots) and CR3 (4 slots) versions. UR1 and UR2 can also be used as expansion racks. The power supply and the CPU also occupy slots in the racks, possibly even two or more per module. If necessary, the number of slots available can be increased using expansion racks: UR1 and ER1 have 18 slots each, UR2 and ER2 have 9 each.

The IM 460-1 and IM 461-1 interface modules make it possible to have one expansion rack per

interface up to 1.5 meters from the central rack, including the 5 V supply voltage. In addition, as many as four expansion racks can be operated up to 5 meters away using IM 460-0 and IM 461-0 interface modules. And finally, IM 460-3 and IM 461-3 or IM 460-4 and 461-4 interface modules can be used to operate as many as four expansion racks at a distance of up to 100 or 600 meters away.

A maximum of 21 expansion racks can be connected to a central rack. To distinguish between racks, you set the number of the rack on the coding switch of the receiving IM.

The backplane bus consists of a parallel P bus and a serial K bus. Expansion racks ER1 and ER2 are designed for “simple” signal modules which generate no hardware interrupts, do not have to be supplied with 24 V voltage via the P bus, require no back-up voltage, and have no K bus connection. The K bus is in racks UR1, UR2 and CR2 either when these racks are used as central racks or expansion racks with the numbers 1 to 6.

Segmented rack

A special feature is the segmented rack CR2. The rack can accommodate two CPUs with a shared power supply while keeping them functionally separate. The two CPUs can exchange data with one another via the K bus, but have completely separate P buses for their own signal modules.

Multicomputing

In an S7-400, as many as 4 specially designed CPUs in a UR central rack can take part in multicomputing. Each module in this station is assigned to only one CPU, both with its address and its interrupts. For further details, see Chapters 20.3.6 “Multicomputing” and 21.7 “Multi-processor Interrupt”.

Connecting SIMATIC S5 modules

The IM 463-2 interface module allows you to connect S5 expansion units (EG 183U, EG 185U, EG 186U as well as ER 701-2 and ER 701-3) to an S7-400, and also allows centralized expansion of the expansion units. An IM 314 in the S5 expansion unit handles the link.

You can operate all analog and digital modules allowed in these expansion units. An S7-400 can accommodate as many as four IM 463-2 interface modules; as many as four S5 expansion units can be connected in a distributed configuration to each of an IM 463-2's two interfaces.

1.1.4 Fault-tolerant SIMATIC

Two designs of SIMATIC S7 fault-tolerant automation systems are available for applications with high fault tolerance demands for machines and processes: software redundancy and S7-400H/FH.

Software redundancy

Using SIMATIC S7-300/400 standard components, you can establish a software-based redundant system with a master station controlling the process and a standby station assuming control in the event of the master failing.

Fault tolerance through software redundancy is suitable for slow processes because transfer to the standby station can require several seconds depending on the configuration of the programmable controllers. The process signals are “frozen” during this time. The standby station then continues operation with the data last valid in the master station.

Redundancy of the input/output modules is implemented with distributed I/O (ET 200M with IM 153-2 interface module for redundant PROFIBUS DP). The software redundancy can be configured with STEP 7 Version 5.2 and higher.

Fault-tolerant SIMATIC S7-400H

The SIMATIC S7-400H is a fault-tolerant programmable controller with redundant configuration comprising two central racks, each with an H CPU and a synchronization module for data comparison via fiber optic cable. Both controllers operate in “hot standby” mode; in the event of a fault, the intact controller assumes operation alone via automatic bumpless transfer. The UR2-H mounting rack with two times nine slots makes it possible to establish a fault-tolerant system in a single mounting rack.

The I/O can have normal availability (single-channel, single-sided configuration) or enhanced availability (single-channel switched configuration with ET 200M). Communication is carried out over a simple or a redundant bus.

The user program is the same as that for a non-redundant controller; the redundancy function is handled exclusively by the hardware and is invisible to the user. The software package required for configuration is included in STEP 7 V5.3 and later. The provided standard libraries *Redundant IO* contain blocks for supporting the redundant I/O.

1.1.5 Safety-related SIMATIC

Failsafe automation systems control processes in which the safe state can be achieved by direct switching off. They are used in plants with increased safety requirements.

The safety functions are located as appropriate in the safety-related user program of a correspondingly designed CPU and in the failsafe input and output modules. An F-CPU complies with the safety requirements up to AK 6 in accordance with DIN V 19250/DIN V VDE 0801, up to SIL 3 in accordance with IEC 61508, and up to Category 4 in accordance with EN 954-1. Safety functions can be executed parallel to a non-safety-related user program in the same CPU.

Safety-related communication over PROFIBUS DP – also over PROFINET IO with S7 Distributed Safety – uses the PROFIsafe bus profile. This permits transmission of safety-related and non-safety-related data on a single bus cable.

Safety Integrated for the manufacturing industry

S7 Distributed Safety is a failsafe automation system for the protection of machines and personnel mainly for applications with machine controls and in the process industry.

CPUs from the SIMATIC S7-300, S7-400 and ET 200S ranges are currently available as F-CPU. The safety-related I/O modules are connected to S7-400 over PROFIBUS DP or PROFINET IO using the safety-related PROFIsafe bus profile. With S7-300 and ET

200S, use of safety-related I/O modules is additionally possible in the central rack.

The hardware configuration and programming of the non-safety-related user program are carried out using the standard applications of STEP 7.

The *SIMATIC S7 Distributed Safety* option package is required to program the safety-related parts of the program. With this option package you can use the F-LAD or F-FBD programming languages to create the blocks which contain the safety-related program. Interfacing to the I/O is carried out using the process image as with the standard program. S7 Distributed Safety also includes a library with TÜV-certified safety blocks. There is an additional library available with F-blocks for press and burner controls.

The safety-related user program can be executed parallel to the standard user program. If an error is detected in the safety-related part of the program, the CPU enters the STOP state.

Safety Integrated for the process industry

S7 F/FH Systems is a failsafe automation system based on S7-400 mainly for applications in the process industry. The safety-related I/O modules are connected over PROFIBUS DP using the safety-related PROFIsafe bus profile.

An S7-400 F-CPU is provided with the safety-related control functions by application of an *S7 F Systems Runtime license*. A non-safety-related user program can be executed parallel to the safety-related plant unit.

In addition to fail-safety, the S7-400FH also provides increased availability. If a detected fault results in a STOP of the master CPU, a reaction-free switch is made to the CPU running in hot standby mode. The *S7 H Systems* option package is additionally required for operation as S7-400FH.

The hardware configuration and programming of the non-safety-related user program are carried out using the standard applications of STEP 7.

The *S7 F Systems* option package is additionally required for programming the safety-related program parts, and additionally the

CFC option package V5.0 SP3 and higher and the S7-SCL option package V5.0 and higher.

The safety-related program is programmed using CFC (Continuous Function Chart). Programmed, safety-related function blocks from the supplied F-library can be called and interconnected in this manner. In addition to functions for programming safety functions, they also contain functions for error detection and response. In the event of faults and failures, this guarantees that the failsafe system is held in a safe state or is transferred to a safe state. If a fault is detected in the safety program, the safety-related part of the plant is switched off, whereas the remaining part can continue to operate.

Failsafe I/O

Failsafe signal modules (F-modules or F-submodules) are required for safety operation. Failsafety is achieved with the integral safety functions and appropriate wiring of the sensors and actuators.

The F-modules can also be used in standard applications with increased diagnostics requirements. The F-modules can be operated in redundant mode to increase the availability both in standard and safety operation with S7 F/FH systems.

The failsafe I/O is available in various versions:

- ▷ The failsafe signal modules of S7-300 design are used in the ET 200M distributed I/O device or – with S7-Distributed Safety – also centrally.
- ▷ Failsafe I/O modules are available for the distributed I/O devices in the designs ET 200S, ET 200pro, and ET 200eco..
- ▷ For the ET 200S and ET 200pro distributed I/O devices, failsafe interface modules are also available as F-CPU.
- ▷ Failsafe DP standard slaves and – with S7-Distributed Safety also IO standard devices – can be used which can handle the PROFIsafe bus profile.

Failsafe CPUs and signal modules are also available in SIPLUS design.

1.1.6 CPU Memory Areas

Figure 1.2 shows the memory areas in the programming device, the CPU and the signal modules which are important for your program.

The programming device contains the *offline data*. These consist of the user program (program code and user data), the system data (e.g. hardware, network and interconnection configurations), and further project-specific data such as symbol tables and comments.

The *online data* consist of the user program and the system data on the CPU, and are accommodated in two areas, namely load memory and work memory. In addition, the system memory is also present here.

The I/O modules contain memories for the signal state of the inputs and outputs.

The CPUs have a slot for a plug-in *memory submodule*. The load memory, or parts thereof, is located here (see “Physical design of CPU memory”, further below). The memory submodule is designed as a memory card (S7-400 CPUs) or as a micro memory card (S7-300 CPUs and ET200 CPUs derived from these). The firmware of the CPU operating system can also be updated using the memory submodule.

Memory card

The memory module for the S7-400 CPUs is the memory card (MC). There are two types of memory card: RAM cards and flash EPROM cards.

If you want to expand load memory only, use a RAM card. A RAM card allows you to modify the entire user program online. This is necessary, for example, when testing and starting up larger programs. RAM memory cards lose their contents when unplugged.

If you want to protect your user program, including configuration data and module parameters, against power failure following testing and starting up even without a backup battery, use a flash EPROM card. In this case, load the entire program offline onto the flash EPROM card with the card plugged into the programming device. With the relevant CPUs, you can also load the program online with the memory card plugged into the CPU.

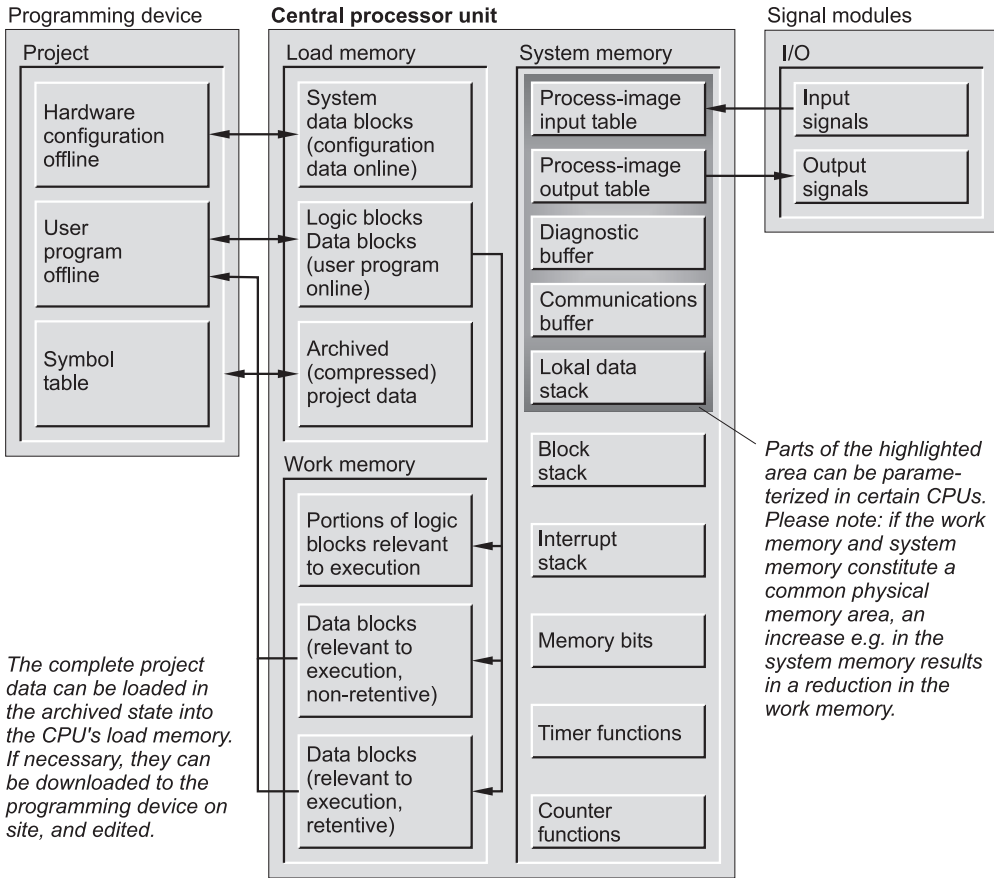


Figure 1.2 CPU Memory Areas

Micro memory card

The memory submodule for the newer S7-300 CPUs is a micro memory card (MMC). The data on the MMC are saved non-volatile, but can be read, written and deleted as with a RAM. This response permits data backup without a battery.

The complete load memory is present on the MMC, meaning that an MMC is always required for operation. The MMC can be used as a portable memory medium for user programs or firmware updates. Using special system functions you can read or write data blocks on the MMC from the user program, for example to read recipes from the MMC or to create a

measured-value archive on the MMC and to provide it with data.

Load memory

The entire user program, including configuration data (system data), is in the load memory. The user program is always initially transferred from the programming device to the load memory, and from there to the work memory. The program in the load memory is not processed as the control program.

With a CPU 300 and a CPU ET 200, the load memory is present completely on the micro memory card. Thus the contents of the load memory are retained even if the CPU is de-energized.

If the load memory with a CPU 400 consists of an integrated RAM or RAM memory card, a backup battery is required in order to keep the user program retentive. With an integrated EEPROM or a plug-in flash EPROM memory card as the load memory, the CPU can be operated without battery backup.

From STEP 7 V5.1 onwards, and with appropriately designed CPUs, you can save the complete project data as a compressed archive file in the load memory (see Chapter 2.2.2 “Managing, Reorganizing and Archiving”).

Work memory

Work memory is designed in the form of high-speed RAM fully integrated in the CPU. The operating system of the CPU copies the program code “relevant to execution” and the user data into the work memory. “Relevant” is a characteristic of the existing objects and does not mean that a particular code block will necessarily be called and executed. The “actual” control program is executed in the work memory.

Depending on the product, the work memory can be designed either as a correlated area or divided according to program and data memories, where the latter can also be divided into retentive and non-retentive memories.

When uploading the user program into the programming device, the blocks are fetched from the load memory, supplemented by the actual values of the data operands from the work memory (further information can be found in Sections 2.6.4 “Loading the User Program into the CPU” and 2.6.5 “Block Handling”).

System memory

System memory contains the addresses (variables) that you access in your program. The addresses are combined into areas (address areas) containing a CPU-specific number of addresses. Addresses may be, for example, inputs used to scan the signal states of momentary-contact switches and limit switches, and outputs that you can use to control contactors and lamps.

The system memory on a CPU contains the following address areas:

- ▷ Inputs (I)
Inputs are an image (“process image”) of the digital input modules.
- ▷ Outputs (Q)
Outputs are an image (“process image”) of the digital output modules.
- ▷ Bit memories (M)
are information stores which are directly accessible from any point in the user program.
- ▷ Timers (T)
Timers are locations used to implement waiting and monitoring times.
- ▷ Counters (Z)
Counters are software-level locations, which can be used for up and down counting.
- ▷ Temporary local data (L)
Locations used as dynamic intermediate buffers during block processing. The temporary local data are located in the L stack, which the CPU occupies dynamically during program execution.

The letters enclosed in parentheses represent the abbreviations to be used for the different addresses when writing programs. You may also assign a symbol to each variable and then use the symbol in place of the address identifier.

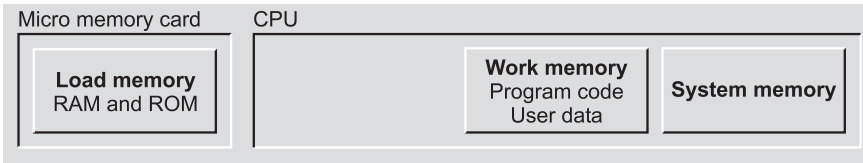
The system memory also contains buffers for communication jobs and system messages (diagnostics buffer). The size of these data buffers, as well as the size of the process image and the L stack, are parameterizable on certain CPUs.

Physical design of CPU memory

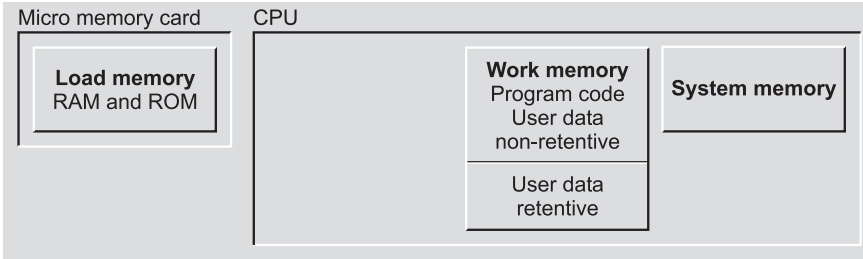
The physical design of the load memory is different for the various types of CPU (Figure 1.3).

A CPU 300 or CPU ET 200 does not have an integrated load memory. A micro memory card containing the load memory must always be inserted to permit operation. The load memory can be written and read like a RAM. The physical design means that the number of write operations is limited (no cyclic writing by user program). You can use the menu command COPY RAM TO ROM to transfer the current values of the data operands from the work memory to the load memory.

S7-300 and ET CPUs without adjustable data retentivity



S7-300 and ET CPUs with adjustable data retentivity



S7-400 CPU

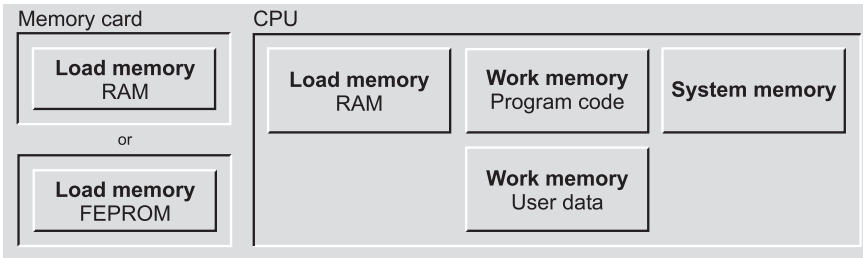


Figure 1.3 Physical Design of CPU Memory

With a CPU 300 with firmware version V2.0.12 or later, the work memory for the user data consists of a retentive part and a non-retentive part. The control program is also present in the non-retentive part.

The integrated RAM load memory in a CPU 400 is designed for small programs or for modification of individual blocks if the load memory is a flash EPROM memory card. If the complete control program is larger than the integrated load memory, you require a RAM memory card for testing. The tested program is then transmitted by the programming device to a flash EPROM memory card which you insert into the CPU for operation.

The work memory of a CPU 400 is divided into two parts: One part saves the program code, the

other the user data. The system and work memories in a CPU 400 constitute one (physical) unit. The system and work memories in the S7-400 CPUs constitute one (physical) unit. If, for example, the size of the process image changes, this has effects on the size of the work memory.

1.2 Distributed I/O

Distributed I/O refers to modules connected via PROFIBUS DP or PROFINET IO. PROFIBUS DP uses the PROFIBUS subnetwork for data transmission, PROFINET IO the Industrial Ethernet subnetwork (for further information, see Chapter 1.3.2 "Subnets").

1.2.1 PROFIBUS DP

PROFIBUS DP provides a standardized interface for transferring predominantly binary process data between an “interface module” in the (central) programmable controller and the field devices. This “interface module” is called the DP master and the field devices are the DP slaves.

The DP master and all the slaves it controls form a DP master system. There can be up to 32 stations in one segment and up to 127 stations in the entire network. A DP master can control a number of DP slaves specific to itself. You can also connect programming devices to the PROFIBUS DP network as well as, for example, devices for operator control and monitoring, ET 200 devices, or SIMATIC S5 DP slaves.

DP master system

PROFIBUS DP is usually operated as a “mono master system”, that is, one DP master controls several DP slaves. The DP master is the only master on the bus, with the exception of a temporarily available programming device (diagnostics and service device). The DP master and the DP slaves assigned to it form a DP master system (Figure 1.4).

You can also install several DP master systems on one PROFIBUS subnet (multi master sys-

tem). However, this increases the response time in individual cases because when a DP master has initialized “its” DP slaves, the access rights fall to the next DP master that in turn initializes “its” DP slaves, etc.

You can reduce the response time if a DP master system contains only a few DP slaves. Since it is possible to operate several DP masters in one S7 station, you can distribute the DP slaves of a station over several DP master systems. In multicomputing, every CPU has its own DP master systems.

DP master

The DP master is the active node on the PROFIBUS network. It exchanges cyclic data with “its” DP slaves. A DP master can be

- ▷ A CPU with integral DP master interface or plug-in interface submodule (e.g. CPU 315-2DP, CPU 417)
- ▷ An interface module in conjunction with a CPU (e.g. IM 467)
- ▷ A CP in conjunction with a CPU (e.g. CP 342-5, CP 443-5)

There are “Class 1 masters” for data exchange in process operation and “Class 2 masters” for service and diagnostics (e.g. a programming device).

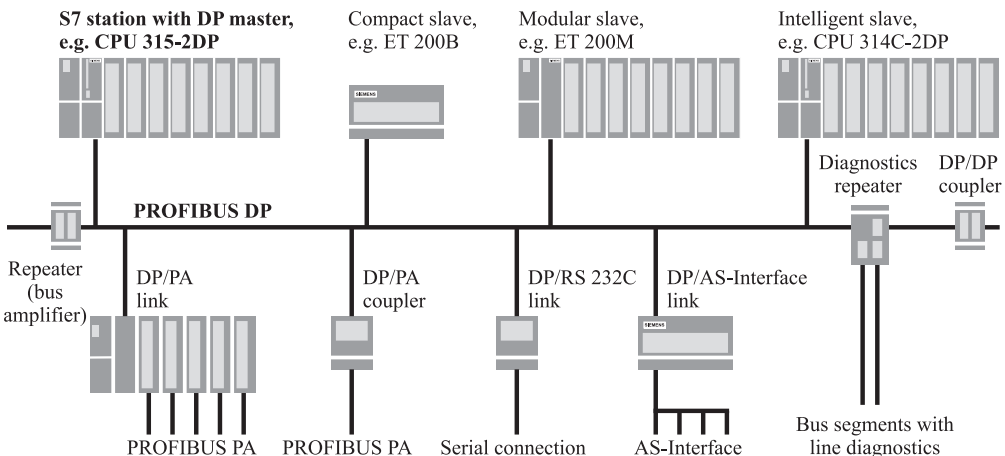


Figure 1.4 Components of a PROFIBUS DP Master System in an RS485 Segment.

DP slaves

The DP slaves are the passive nodes on PROFIBUS. In SIMATIC S7, a distinction is made between

- ▷ Compact DP slaves
They behave like a single module towards the DP master
- ▷ Modular DP slaves
They comprise several modules (submodules)
- ▷ Intelligent DP slaves
They contain a control program that controls the lower-level (own) modules

Compact PROFIBUS DP slaves

Examples of compact DP slaves are the ET 200L, the ET 200R, and the ET 200eco. The bus gateways such as DP/AS-i link behave like a compact slave on PROFIBUS DP.

Modular PROFIBUS DP slaves

Examples of modular DP slaves are the ET 200iSP, the ET 200M, the ET 200S, and the ET 200pro.

Intelligent PROFIBUS DP slaves

Examples of intelligent DP slaves are CPUs with an integral DP (slave) interface, or an S7-300 station with the CP 342-5 communications processor. Equally, an ET 200pro station with the IM 154-8 PN/DP CPU interface module or an ET 200S station with the IM 151-7 CPU interface module can be operated as intelligent DP slaves.

RS 485 repeater

The RS 485 repeater combines two bus segments in a PROFIBUS subnetwork. As a result, the number of stations and the expansion of the subnetwork can be increased.

The repeater provides signal regeneration and electrical isolation. It can be operated at transmission rates up to 12 Mbit/s, including 45.45 kbit/s for PROFIBUS PA.

The RS 485 is not configured; it need only be considered when calculating the bus parameters.

Diagnostics repeater

Using a diagnostics repeater, you can determine the topology and carry out diagnostics in a PROFIBUS segment (RS 485 copper cable) during runtime. The diagnostics repeater provides signal regeneration and electrical isolation of the connected segments. The maximum segment length is 100 m in each case; the transmission rate can be between 9.6 kbit/s and 12 Mbit/s.

The diagnostics repeater has connections for three bus segments. The cable from the DP master is connected to the infeed terminals of bus segment DP1. The two other connections DP2 and DP3 contain the test circuits for determination of the topology and line diagnostics on the connected bus segments. Up to 9 further diagnostics repeaters can be connected in series.

The diagnostics repeater is handled like a DP slave in the master system. In the event of a fault, it sends the determined diagnostics data to the DP master. These are the topology of the bus segment (stations and cable lengths), the contents of the segment diagnostics buffers (last ten events with fault information, location and cause) and the statistics data (statement on quality of bus system). In addition, the diagnostics repeater provides monitoring functions for isochrone mode.

The diagnostics data can be fetched and also graphically displayed by a programming device with STEP 7 V5.2 or later. Line diagnostics is triggered from the user program by the system function SFC 103 DP_TOPOL, and read using SFC 59 RD_REC or SFB 52 RDREC. In order to set the clock on the diagnostics repeater, you read the CPU time using the system function SFC 1 READ_CLK and transmit it using SFC 58 WR_REC or SFB 53 WRREC.

The diagnostics repeater is configured and parameterized using STEP 7. A GSD file is available for operation on non-SIMATIC masters.

1.2.2 PROFINET IO

PROFINET IO offers a standardized interface for transmission of mainly binary process data between an "interface module" in the (central)

programmable controller and the field devices using Industrial Ethernet. This “interface module” is referred to as the IO controller and the field devices as IO devices. The IO controller with all the IO devices controlled by it constitute a PROFINET IO system.

PROFINET IO system

A PROFINET IO system comprises the IO controller in the central station and the IO devices (field devices) assigned to it. The Industrial Ethernet subnet connecting them can also be shared by other stations and applications (Figure 1.5).

IO controller

The IO controller is the active station on the PROFINET. It exchanges data cyclically with “its” IO devices. An IO controller can be:

- ▷ A CPU with integral PROFINET interface (e.g. CPU 317-2PN/DP)
- ▷ A CP module in conjunction with a CPU (e.g. CP 343-1)

IO device

The IO devices are the passive stations on the PROFINET. In SIMATIC S7, a distinction is made between:

- ▷ Compact IO devices
These behave like a single module with regard to the IO controller

- ▷ Modular IO devices
These comprise several modules (submodules)
- ▷ Intelligent IO devices
These contain a control program that controls the lower-level (own) modules.

Compact PROFINET IO devices

An example of a compact IO device is the ET 200eco. Bus gateways such as the IE/AS-i Link PN IO also behave like a compact slave on the PROFINET IO.

Modular PROFINET IO devices

Examples of modular IO devices are the ET 200M, the ET 200S, and the ET 200pro.

Intelligent PROFINET IO devices

Intelligent IO devices are, for example, CPUs with integrated PN interface. Equally, an ET 200pro station with the IM 154-8 PN/DP CPU interface module or an ET 200S station with the IM 151-8 PN/DP CPU interface module can be operated as intelligent IO devices.

IO supervisor

IO supervisors are devices for parameterization, startup, diagnostics, and human machine interfacing, e.g. programming devices or HMI devices.

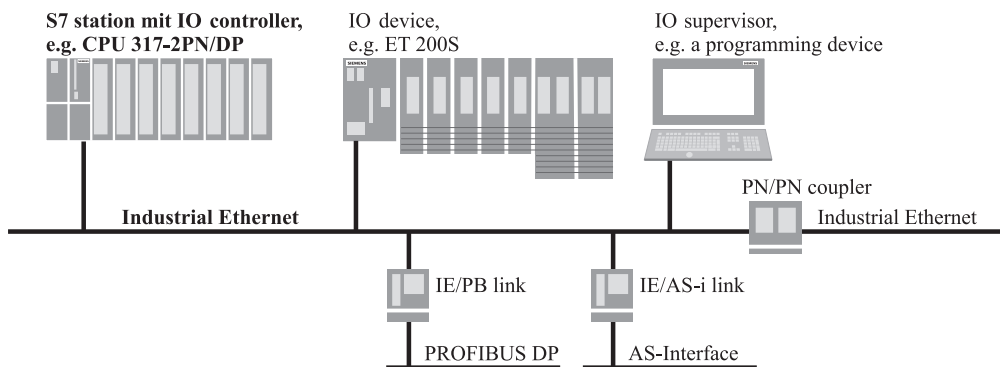


Figure 1.5 Components of a PROFINET IO system

1.2.3 Actuator/Sensor Interface

The Actuator/Sensor interface (AS-i) is a networking system for the lowest process level in automation plants in accordance with the international standard EN 50295. An AS-i master controls up to 62 AS-i slaves via a 2-wire AS-i cable that carries both the control signals and the supply voltage. (Figure 1.6).

One AS-i segment can be up to 100 m in length; in combination with repeaters and extension plugs, a maximum expansion of 600 m can be achieved.

With the *ASISafe* safety concept, you can connect safety sensors such as emergency-off switches, door contact switches, or safety light arrays directly to the AS-i network up to Category 4 in accordance with EN 954-1 or SIL3 in accordance with IEC 61508. This requires safe AS-i slaves for connecting the safety sensors and a safety monitor that combines the safe inputs with parameterizable logic and ensures safe shutdown.

AS-i master

Standard AS-i masters can control up to 31 standard AS-i slaves with a maximum cycle time of 5 ms. In the case of extended AS-i masters, the quantity structure increases to a maxi-

mum of 62 AS-i slaves with an extended address area with a maximum cycle time of 10 ms. Slaves with an extended address area occupy one address in pairs; if standard slaves are operated on an extended master, they each occupy one address.

The **AS-i master CP 343-2** is used in an S7-300 station or in an ET200M station. It supports the following AS-i slaves:

- ▷ Standard slaves
- ▷ Slaves with extended addressing mode (A/B slaves)
- ▷ Analog slaves to slave profile 7.3 or 7.4

In *standard mode*, the CP 343-2 behaves like an I/O module: It occupies 16 input bytes and 16 output bytes in the analog address area (from 128 upwards). Up to 31 standard slaves or 62 A/B slaves (slaves with extended address area) can be operated on the CP 343-2. The AS-i slaves are parameterized with default values stored in the CP.

In *extended mode*, the full range of functions in accordance with the AS-i master specification is available. If you use the FC block supplied, master calls can be made from the user program in addition to standard mode (transfer of parameters during operation, checking of the desired/actual configuration, test and diagnostics).

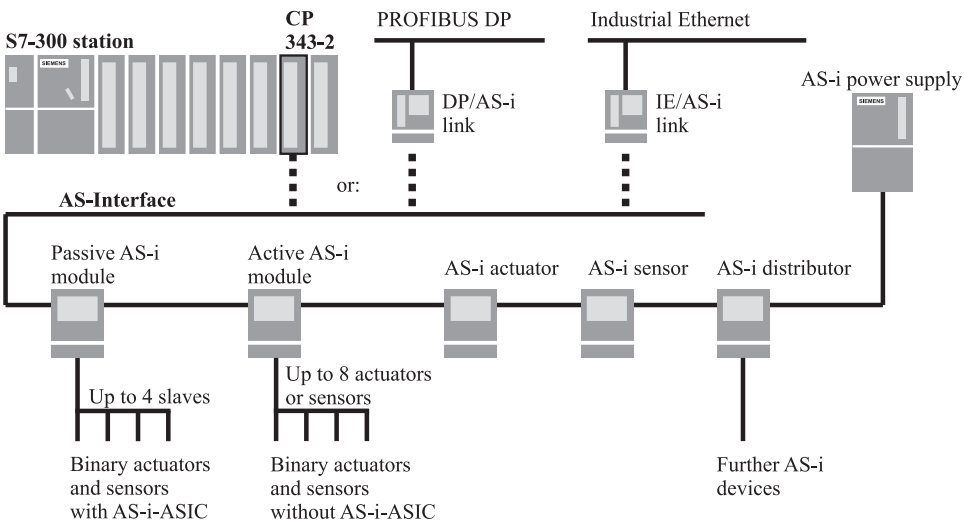


Figure 1.6 Connecting the AS-i bus system to SIMATIC S7

There are two methods of linking PROFIBUS DP and PROFIBUS PA:

- ▷ DP/PA coupler, when PROFIBUS DP can be operated at 45.45 kbit/s
- ▷ DP/PA link which converts the data transfer rates of PROFIBUS DP to the data transfer rate of PROFIBUS PA

The **DP/PA coupler** enables connection of PA field devices to PROFIBUS DP. On PROFIBUS DP, the DP/PA coupler is a DP slave that is operated at 45.45 kbit/s. Up to 31 PA field devices can be connected to one DP/PA coupler. The field devices form a PROFIBUS PA segment with a data transfer rate of 31.25 kbit/s. All PROFIBUS PA segments together form a shared PROFIBUS PA bus system.

The DP/PA coupler is available in two versions: a non-Ex version with up to 400 mA output current and an Ex version with up to 100 mA output current.

The **DP/PA link** enables the connection of PA field devices to PROFIBUS DP with data transfer rates between 9.6 kbit/s and 12 Mbit/s. A DP/PA link comprises an IM 157 interface module and up to 5 DP/PA couplers that are connected to each other via SIMATIC S7 bus connectors. It maps the bus system consisting of all PROFIBUS PA segments to a PROFIBUS DP slave. A maximum of 31 PA field devices can be connected per DP/PA link.

SIMATIC PDM (Process Device Manager, previously SIPROM) is a cross-vendor tool for parameterization, startup and diagnostics of intelligent field devices with PROFIBUS PA or HART functionality. The DDL (Device Description Language) is available for parameterizing HART transducers (Highway Addressable Remote Transducers).

From STEP 7 V5.1 SP3, the control technology modules are parameterized with the Hardware Configuration; you must then no longer use SIMATIC PDM.

Connecting PROFIBUS DP to the AS-Interface

A **DP/AS-Interface link** enables the connection of PROFIBUS DP to the AS-Interface. On PRO-

FIBUS DP, the link is a modular DP slave with a data transfer rate of up to 12 Mbit/s in degree of protection IP 20. On the AS-Interface, it is an AS-i master that controls the AS-i slaves. The link is available in the versions *DP/AS-i Link 20E* and *DP/AS-i Link Advanced*. The following AS-i slaves can be controlled:

- ▷ Standard slaves, AS-i analog slaves
- ▷ Slaves with extended addressing mode (A/B slaves)
- ▷ Slaves with data transfer mechanisms in accordance with AS-i specification V3.0 (DP/AS-i Link Advanced)

Connection of PROFIBUS DP to a serial interface

The **PROFIBUS DP/RS 232C link** is a converter between an RS 232C (V.24) interface and PROFIBUS DP. Devices with an RS 232C interface can be connected to PROFIBUS DP with the DP/RS 232C link. The DP/RS 232C link supports the procedures 3964R and free ASCII protocol.

The PROFIBUS DP/RS 232C link is connected to the device via a point-to-point connection. Conversion to the PROFIBUS DP protocol takes place in the PROFIBUS DP/RS 232C link. The data is transferred consistently in both directions. Up to 224 bytes of user data can be transferred per message frame.

The data transfer rate on PROFIBUS DP can be up to 12 Mbit/s; RS 232C can be operated at up to 38.4 kbit/s with no parity, even or odd parity, 8 data bits, and 1 stop bit.

Connecting two PROFINET subnets

With the **PN/PN coupler**, you interconnect two Ethernet subnets in order to exchange data between the IO controllers of both subnets. There is galvanic isolation between the subnets.

The PN/PN coupler is a 120-mm-wide module that is installed on a DIN rail. The subnets are connected using RJ45 connectors. Two connections with internal switch function are available for each subnet.

From the viewpoint of the relevant IO controller, the PN/PN coupler is an IO device in its own PROFINET IO system. Both IO devices

are linked by a data transfer area with 256 input bytes and 256 output bytes, divisible into a maximum of 16 areas. Input areas in one subnet must correspond to output areas in another.

The PN/PN coupler is configured and parameterized with STEP 7. A GSDML file is available for other configuring tools.

Connection of PROFINET IO to PROFIBUS DP

You can connect the Industrial Ethernet subnetworks and PROFIBUS using the **IE/PB link PNIO**. If you use PROFINET IO, the IE/PB link PNIO takes over the role of a proxy for the DP slaves on the PROFIBUS. An IO controller can access DP slaves just like IO devices using the IE/PB link. In standard mode, the IE/PB link is transparent for PG/OP communications and S7 routing between subnetworks.

The IE/PB link PNIO is a double-width module of S7-300 design. The IE/PB link is connected to Industrial Ethernet via an 8-contact RJ45 socket, and to PROFIBUS via a 9-contact SUB-D socket.

The IE/PB link is configured using STEP 7 as an IO device to which a DP master system is connected. When switching on, the subordinate DP slaves are also provided with the configuration data from the IO controller.

Please note that limitations exist on the PROFIBUS DP following an IE/PB link. For example, you cannot connect a DP/PA link, the DP segment does not have CIR capability, and isochrone mode cannot be configured.

Connecting PROFINET IO to the AS-Interface

An **IE/AS-i link** enables the connection of PROFINET IO to the AS-Interface. On PROFINET IO, the link is an IO device. On PROFIBUS, the link is an IO device. On the AS-Interface, it is the AS-i master that controls the AS-i slaves. The IO controller can access the individual binary and analog values of the AS-i slaves directly.

Connection to PROFINET is made via two RJ45 connectors with internal switch function. The AS-Interface bus is connected to 4-pin plug-in screw-type contacts.

The link is available in the versions single master and double master (in accordance with AS-Interface specification V3.0) for the connection of up to 62 AS-i slaves in each case and integral analog value transfer. The following AS-i slaves can be controlled:

- ▷ Standard slaves, AS-i analog slaves
- ▷ Slaves with extended address area (extended addressing mode, A/B slaves)
- ▷ Slaves with data transfer mechanisms in accordance with AS-i specification V3.0

The IE/AS-i link is configured and parameterized with STEP 7. A GSDML file is available for other configuring tools.

1.3 Communications

Communications – data exchange between programmable modules – is an integral component of SIMATIC S7. Almost all communications functions are handled via the operating system. You can exchange data without any additional hardware and with just one connecting cable between the two CPUs. If you use CP modules, you can achieve powerful network links and the facility of linking to non-Siemens systems.

SIMATIC NET is the umbrella term for SIMATIC communications. It represents information exchange between programmable controllers and between programmable controllers and human machine interface devices. There are various communications paths available depending on performance requirements.

1.3.1 Introduction

The most significant communications objects are initially SIMATIC stations or non-Siemens devices between which you want to exchange data. You require modules with communications capability here. With SIMATIC S7, all CPUs have an MPI interface over which they can handle communications.

In addition, there are communications processors (CPs) available that enable data exchange at higher throughput rates and with different protocols. You must link these modules via net-

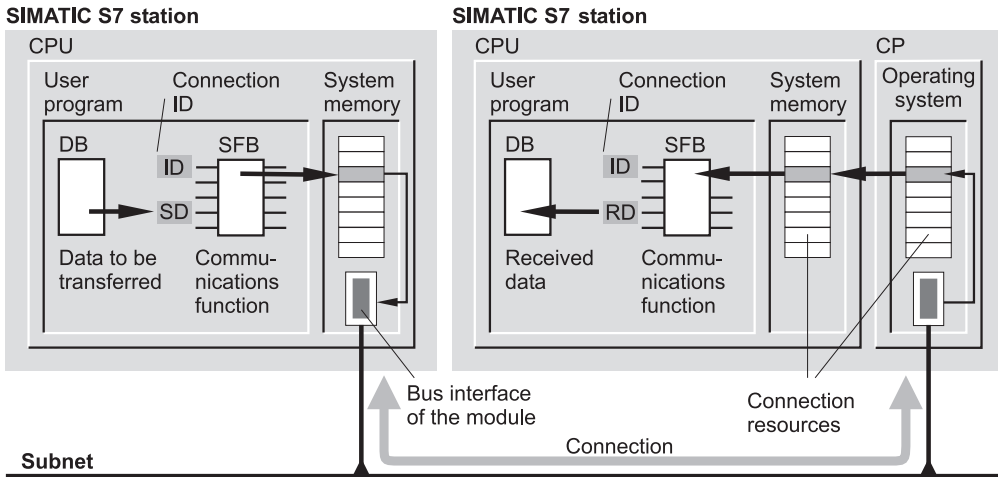


Figure 1.8 Data Exchange Between Two SIMATIC S7 Stations

works. A network is the hardware connection between communication nodes.

Data is exchanged via a “connection” in accordance with a specific execution plan (“communications service”) which is based, among other things, on a specific coordination procedure (“protocol”). S7 connection is the standard between S7 modules with communications capability, for example.

Using an S7 connection, Figure 1.8 shows the objects involved in communication between two stations. The user program of the left-hand station contains the data to be transmitted in a data block (DB). The communications function in the example is a system function block (SFB). Assign the parameter RD with a pointer to the data to be sent, and trigger the transmission from the program. The communications function is additionally assigned a connection ID with which the used connection is specified. The connection occupies a connection resource in the CPU’s system memory. The data are transmitted e.g. to a CP module in another station via the module’s bus interface. Connection resources are used in both the CP module and CPU. Because of the connection ID (and the configured connection path) the communications function in the receiver station “recognizes” the data addressed to it, and writes them into the data block of the user program by means of the pointer in parameter RD.

Network

A network is a connection between several devices for the purpose of communication. It comprises one or more identical or different subnets linked together.

Subnet

In a subnet, all the communications nodes are linked via a hardware connection with uniform physical characteristics and transmission parameters, such as the data transfer rate, and they exchange data via a shared transmission procedure. SIMATIC recognizes MPI, PROFIBUS, Industrial Ethernet and point-to-point connection (PTP) as subnets.

Communications service

A communications service determines how the data are exchanged between communications nodes and how the data are to be handled. It is based on a protocol that describes, amongst other things, the coordination procedure between the communications nodes.

The services mostly used with SIMATIC are: PG communications, OP communications, S7 basic communications, S7 communications, global data communications, PtP communications, S5-compatible communications (SEND/RECEIVE interface).

Connection

A connection defines the communications relationships between two communications nodes. It is the logical assignment of two nodes for executing a specific communications service and also contains special properties such as the connection type (dynamic, static) and how it is established.

SIMATIC recognizes the following connection types: S7 connection, S7 connection (fault-tolerant), point-to-point connection, FMS and FDL connections, ISO transport connection, ISO-on-TCP and TCP connections, UDP connection and e-mail connection.

Communications functions

The communications functions are the user program's interface to the communications service. For SIMATIC S7-internal communications, the communications functions are integrated in the operating system of the CPU and they are called via system blocks. Loadable blocks are available for communication with non-Siemens devices via communications processors.

Overview of communications objects

Table 1.1 shows the relationships between subnets, modules with communications capability and communications services. In addition to the communications services shown, PG/OP communications is also possible via MPI, PROFIBUS and Industrial Ethernet subnets.

1.3.2 Subnets

Subnets are communications paths with the same physical characteristics and the same communications procedure. Subnets are the central objects for communication in the SIMATIC Manager.

The subnets differ in their performance capability:

- ▷ MPI
Low-cost method of networking a few SIMATIC devices with small data volumes.
- ▷ PROFIBUS
High-speed exchange of small and mid-

range volumes of data, used primarily with distributed I/O.

- ▷ Industrial Ethernet
Communications between computers and programmable controllers for high-speed exchange of large volumes of data, also used with distributed I/Os (PROFINET IO).
- ▷ Point-to-point (PTP)
Serial link between two communications partners with special protocols.

From STEP 7 V5, you can use a programming device to reach SIMATIC S7 stations via subnets, for the purposes of, say, programming or parameterizing. The gateways between the subnets must be located in an S7 station with “routing capability”.

MPI

Every CPU with SIMATIC S7 has an “interface with multipoint capability” (multipoint interface, or MPI). It enables establishment of subnets in which CPUs, human machine interface devices and programming devices can exchange data with each other. Data exchange is handled via a Siemens proprietary protocol.

The maximum number of nodes on the MPI network is 32. Each node has access to the bus for a specific length of time and may send data frames. After this time, it passes the access rights to the next node (“token passing” access procedure).

As transmission medium, MPI uses either a shielded twisted-pair cable or a glass or plastic fiber-optic cable. The maximum cable length in a bus segment with non-electrically-isolated interfaces is up to 50 m depending on the transmission rate, and up to 1000 m with electrically isolated interfaces. This can be increased by inserting RS485 repeaters (up to 1100 m) or optical link modules (up to > 100 km). The data transfer rate is usually 187.5 kbit/s.

Over an MPI subnet, you can exchange data between CPUs with global data communications, station-external S7 basic communications or S7 communications. No additional modules are required.

Table 1.1 Communications Objects

Subnet	Modules	Communications Service, Connection	Configuring, Interface
MPI	All CPUs	Global data communications	GD table
		Station-internal S7 basic communications	SFC calls
		S7 communications	Connection table, FB/SFB calls
PROFI-BUS	CPUs with DP interface	PROFIBUS DP (DP master or DP slave)	Hardware configuration, SFB/SFC calls, inputs/outputs
		Station-internal S7 basic communications	SFC calls
	IM 467	PROFIBUS DP (DP master)	Hardware configuration, SFB/SFC calls, inputs/outputs
		Station-internal S7 basic communications	SFC calls
	CP 342-5 CP 443-5 Extended	CP 342-5: PROFIBUS DP V0 CP 433-5 Ext.: PROFIBUS DP V1 (DP master or DP slave)	Hardware configuration, SFB/SFC calls, inputs/outputs
		Station-internal S7 basic communications	SFC calls
		S7 communications	Connection table, FB/SFB calls
		S5-compatible communications	NCM, connection table, SEND/RECEIVE
	CP 343-5 CP 443-5 Basic	Station-internal S7 basic communications	SFC calls
		S7 communications	Connection table, FB/SFB calls
		S5-compatible communications	NCM, connection table, SEND/RECEIVE
		PROFIBUS FMS	NCM, connection table, FMS interface
Industrial Ethernet	CPUs with PN interface	PROFINET IO (IO controller)	Hardware configuration, SFB/SFC calls, inputs/outputs
		IE communications	FB calls
	CP 343-1 Lean CP 343-1 CP 443-1	S7 communications	Connection table, FB/SFB calls
		S5-compatible communications Transport protocols TCP/IP and UDP, also ISO with CP 443-1	NCM, connection table, SEND/RECEIVE
	CP 343-1 IT CP 443-1 Advanced CP 443-1 IT	S7 communications	Connection table, FB/SFB calls
		S5-compatible communications Transport protocols TCP/IP and UDP, also ISO with CP 443-1	NCM, connection table, SEND/RECEIVE
		IT communications (HTTP, FTP, E-mail)	NCM, connection table, SEND/RECEIVE
	CP 343-1 PN	S7 communications	Connection table, FB/SFB calls
		S5-compatible communications Transport protocols TCP and UDP	NCM, connection table, SEND/RECEIVE

NCM is the configuring software for CP modules (integrated in STEP 7 V5.2 and later)

PROFIBUS

PROFIBUS stands for “Process Field Bus” and is a vendor-independent standard complying with IEC 61158/EN 50170 for universal automation (PROFIBUS DP and PROFIBUS FMS) and for process automation according to IEC 61158-2 (PROFIBUS PA).

The maximum number of nodes in a PROFIBUS network is 127, where the network is divided into segments with up to 32 nodes. A distinction is made between active and passive nodes. An active node receives access rights to the bus for a specific length of time and may send data frames. After this time, it passes the access rights to the next node (“token passing” access procedure). If passive nodes (slaves) are assigned to an active node (master), the master executes data exchange with the slaves assigned to it while it is in possession of the access rights. A passive node does not receive access rights.

The PROFIBUS network can also be physically designed as an electrical network, optical network or wireless coupling with various transmission rates. The length of a segment depends on the transmission rate. The electrical network can be configured with a linear or tree topology. It uses a shielded, twisted two-wire cable (RS485 interface). The transmission rate can be adjusted in steps from 9.6 kbit/s to 12 Mbit/s (31.25 kbit/s with PROFIBUS PA).

The optical network uses either plastic, PCF or glass fiber-optic cables. It is suitable for large distances, provides electrical isolation, and is insensitive to electromagnetic interferences. The transmission rate can be adjusted in steps from 9.6 kbit/s to 12 Mbit/s. When using optical link modules (OLMs), designs are possible with linear, ring or star topologies. An OLM also provides the connection between electrical and optical networks with a mixed design. A cost-optimized version is the design as a linear structure with integral interface and optical bus terminal (OBT).

Using the PROFIBUS infrared link module (ILM), single or several PROFIBUS slaves or segments can be provided with a wireless connection to PROFIBUS slaves. The maximum transmission rate of 1.5 Mbit/s and the maxi-

mum range of 15 m means that communication is possible with moving parts.

You implement connection of distributed I/O via a PROFIBUS subnetwork; the relevant PROFIBUS DP communications service is implicit. You can use either CPUs with integral or plug-in DP master, or the relevant CPs. You can also operate station-internal S7 basic communications or S7 communications via this network.

You can transfer data with PROFIBUS FMS and PROFIBUS FDL using the relevant CPs. There are loadable blocks (FMS interface or SEND/RECEIVE interface) available as the interface to the user program).

Industrial Ethernet

Industrial Ethernet is the subnet for connecting computers and programmable controllers, with the focus on the industrial area, defined by the international standard IEEE 802.3/802.3u. The standards IEEE 802.11 a/b/g/h define the connection to wireless local area networks (WLAN) and Industrial Wireless LANs (IWLAN).

The number of stations networked using Industrial Ethernet is unlimited; up to 1024 stations are permissible per segment. Before accessing, each node checks to see if another node is currently transmitting. If this is the case, the node waits for a random time before attempting another access (CSMA/CD access procedure). All nodes have equal access rights.

The physical connections on Industrial Ethernet consist of point-to-point connections between communication nodes. Each node is connected with precisely one partner. To enable several nodes to communicate with each other, they are connected to a “distributor” (switch or hub) that has several connections.

A *switch* is an active bus element that regenerates signals, prioritizes them, and distributes them only to those devices that are connected to it. A *hub* adjusts to the lowest data transfer rate at the connections, and forwards all signals unprioritized to all connected devices.

The network can be configured as a linear, star, tree or ring topology. The data transfer rates are

10 Mbit/s, 100 Mbit/s (Fast Ethernet) or 1000 Mbit/s (Gigabit Ethernet, not on PROFINET).

Industrial Ethernet can be physically designed as an electrical network, optical network or wireless network. FastConnect Twisted Pairs (FC TP) with RJ45 connections, or Industrial Twisted Pairs (ITP) with sub-D connections are available for implementing the electrical cabling. Fiber optic (FO) cabling can consist of glass fiber, PCF or POF. It offers galvanic isolation, is impervious to electromagnetic influences, and is suitable for long distances. Wireless transmission uses the frequencies 2.4 GHz and 5 GHz with data transfer rates up to 54 Mbit/s (depending on the national approvals).

You can exchange data with S7 and IE communications via Industrial Ethernet and utilize the S7 functions. With appropriately designed modules, you can also establish ISO transport connections, ISO-on-TCP connections, TCP, UDP and e-mail connections.

PROFINET

PROFINET is the open Industrial Ethernet standard of PROFIBUS International (PI). PROFINET uses the Industrial Ethernet subnet as the physical medium for data transmission, taking into account the requirements of industrial automation. For example, PROFINET offers a real-time (RT) response for communication with field devices, and isochronous real-time (IRT) transmission for motion control applications. Compatibility with TCP/IP and the IT standards of Industrial Ethernet are retained.

Siemens applies PROFINET to two automation concepts:

- ▷ *Component Based Automation (CBA)* uses PROFINET for communication between control units as components in distributed systems. The configuration tool is SIMATIC iMap.
- ▷ *PROFINET IO* uses PROFINET to transmit data to and from field devices (distributed I/O). The configuration tool is SIMATIC STEP 7.

Point-to-point connection

A point-to-point connection (PTP) enables data exchange via a serial link. A point-to-point connection is handled by the SIMATIC Manager as a subnet and configured similarly.

The transmission medium is an electrical cable with interface-dependent assignment. RS 232C (V.24), 20 mA (TTY) and RS 422/485 are available as interfaces. The data transfer rate is in the range 300 bits/s to 19.2 kbit/s with a 20 mA interface or 76.8 kbit/s with RS 232C and RS 422/485. The cable length depends on the physical interface and the data transfer rate; it is 10 m with RS 232C, 1000 m with a 20 mA interface at 9.6 kbit/s and 1200 with RS 422/485 at 19.2 kbit/s.

3964 (R), RK 512, printer drivers and an ASCII driver are available as protocols (procedures), the latter enabling definition of a user-specific procedure.

AS-Interface

The AS-Interface (actuator/sensor interface, AS-i) networks the appropriately designed binary sensors and actuators in accordance with the AS-Interface specification IEC TG 178. The AS-Interface does not appear in the SIMATIC Manager as a subnet; only the AS-I master is configured with the hardware configuration or with the network configuration.

The transmission medium is an unshielded twisted-pair cable that supplies the actuators and sensors with both data and power (power supply required). A network range of up to 600 m can be achieved with repeaters and extension plugs. The data transfer rate is set at 167 kbit/s.

A master controls up to 62 slaves through cyclic scanning and so guarantees a defined response time.

1.3.3 Communications Services

Data exchange over the subnets is controlled by different communications services – depending on the connection selected. The services are provided by the CPU or CP modules. In addition to communications with field devices (PROFIBUS DP, PROFIBUS PA and PROFINET IO, see Chapter 1.2.1 “PROFIBUS DP”

and 1.2.2 “PROFINET IO”), the services listed below are available depending on the module used.

PG communications

PG communications is used to exchange data between an engineering station and a SIMATIC station. It is used, for example, by a programming device in online mode to execute the functions “Monitor variables” or “Read diagnostics buffer” or to load user programs. The communications functions required for PG communications are integrated in the operating system of the SIMATIC modules. PG communications can be executed over the MPI, PROFIBUS and Industrial Ethernet subnets. By applying S7 routing, the PG communications can also be used beyond subnets.

OP communications

OP communications is used to exchange data between an operator station and a SIMATIC station. For example, it is used by an HMI device for operation and monitoring, or to read and write variables. The communications functions required for OP communications are integrated in the operating system of the SIMATIC modules. OP communications can be executed over the MPI, PROFIBUS and Industrial Ethernet subnets.

S7 basic communications

S7 basic communications is an event-controlled service for exchanging small volumes of data between a CPU and a module in the same SIMATIC station (“station-internal”) or between a CPU and a module in a different SIMATIC station (“station-external”). The connections are established dynamically when required. The communications functions required for the S7 basic communications are integrated in the operating system of the CPU, e.g. you trigger the data transfer in the user program by means of system functions SFC. Station-internal S7 basic communications is executed over PROFIBUS, station-external over MPI.

S7 communications

S7 communications is an event-controlled service for exchanging larger volumes of data between CPU modules with control and monitoring functions. The connections are static, and are programmed using STEP 7. The communications functions required for S7 communications are either integrated in the operating system of the CPU (system function blocks SFB) or they are loadable function blocks (FB). S7 communications can be executed over the MPI, PROFIBUS and Industrial Ethernet subnets.

IE communication

With “Open communication via Industrial Ethernet” (IE communication for short), you transfer data between two devices connected to the Ethernet subnet. Communication can be implemented using the protocols TCP native in accordance with RFC 793, ISO-on-TCP in accordance with RFC 1006, or UDP in accordance with RFC 768. The communication functions are loadable function blocks (FBs) contained in the *Standard Library* of STEP 7 under *Communication Blocks*. The function blocks are called in the main program and they control connection buildup and cleardown as well as data transfer.

Global data communications

Global data communications enables exchange of small volumes of data between several CPUs without additional programming overhead in the user program. Transfer can be cyclic or event-driven. The communications functions required are integrated in the operating system of the CPU. Global data communications is only possible over the MPI bus or the K bus.

PtP communications

PtP communications (point-to-point connection) transfers data over a serial interface, e.g. between a SIMATIC station and a printer. The communications functions required are integrated in the operating system, e.g. as system function blocks SFB. Data exchange is possible using various transfer procedures.

S5-compatible communications

S5-compatible communications is an event-controlled service for data transfer between SIMATIC stations and non-SIMATIC stations. The connections are static, and are programmed using STEP 7. The communications functions are usually loadable functions FC with which you can control the transfer from the user program. Data are sent and received over the SEND/RECEIVE interface, and data can be fetched and written over the FETCH/WRITE interface (S7 is a passive partner). S5-compatible communications can be implemented with Industrial Ethernet using the TCP, ISO on TCP, ISO transport and UDP connections, and with PROFIBUS using FDL.

Standard communications

Standard communications uses standardized, vendor-independent protocols for data transfer.

PROFIBUS FMS (Fieldbus Message Specification) provides services for the program-based, device-independent transfer of structured variables (FMS variables) in accordance with EN 50170 Volume 2. Data exchange is carried out using static FMS connections over a PROFIBUS subnet. The communications functions are loadable function blocks FB with which you control the transfer from the user program.

Using an IT communications processor, a SIMATIC station is linked to the **IT communications**. Transfer over Industrial Ethernet comprises PG/OP/S7 communications and S5-compatible communications (SEND/RECEIVE) with the ISO, TCP/IP and UDP transport protocols. It is additionally possible to use SMTP (Simple Mail Transfer Protocol) for e-mails, HTTP (Hyper Text Transfer Protocol) for access using Web browsers, and FTP (File Transfer Protocol) for program-controlled data exchange with devices from different operating systems.

1.3.4 Connections

A connection is either dynamic or static depending on the communications service se-

lected. Dynamic connections are not configured; their buildup or clear-down is event-driven (“Communications via non-configured connections”). There can only ever be one non-configured connection to a communications partner.

Static connections are configured in the connection table; they are built up at startup and remain throughout the entire program execution (“communications via configured connections”). Several connections can be established in parallel to one communications partner. You use a “Connection type” to select the desired communications service in the network configuration (see Chapter 2.4 “Configuring the Network”).

You do not need to configure connections with the network configuration for global data communications and PROFIBUS DP or for S7 basic communications in the case of S7 functions. You define the communications partners for global data communications in the global data table; in the case of PROFIBUS DP and S7 basic communications, you define the partners via the node addresses.

Connection resources

Each connection requires connection resources on the participating communications partner for the end point of the connection or the transition point in a CP module. If, for example, S7 functions are executed via a bus interface of the CPU, a connection is assigned in the CPU; the same functions via the MPI interface of the CP occupy one connection in the CP and one connection in the CPU.

Each CPU has a specific number of possible connections. Limitations and rules exist regarding the use of connection resources. For example, not every connection resource can be used for every type of connection. One connection is reserved for a programming device and one connection for an OP (these cannot be used for any other purpose).

Connection resources are also required temporarily for the “non-configured connections” in S7 basic communications.

1.4 Module Addresses

1.4.1 Signal Path

When you wire your machine or plant, you determine which signals are connected where on the programmable controller (Figure 1.9).

An input signal, for example the signal from momentary-contact switch +HP01-S10, the one for “Switch motor on”, is run to an input module, where it is connected to a specific terminal. This terminal has an “address” called the I/O address (for instance byte 5, bit 2).

Before every program execution start, the CPU then automatically copies the signal to the process input image, where it is then accessed as an

“input” address (I 5.2, for example). The expression “I 5.2” is the absolute address.

You can now give this input a name by assigning an alphanumeric symbol corresponding to this input signal (such as “Switch motor on”) to the absolute address in the symbol table. The expression “Switch motor on” is the symbolic address.

1.4.2 Slot Address

Every slot has a fixed address in the programmable controller (an S7 station). This slot address consists of the number of the mounting rack and the number of the slot. A module is uniquely described using the slot address (“geographical address”).

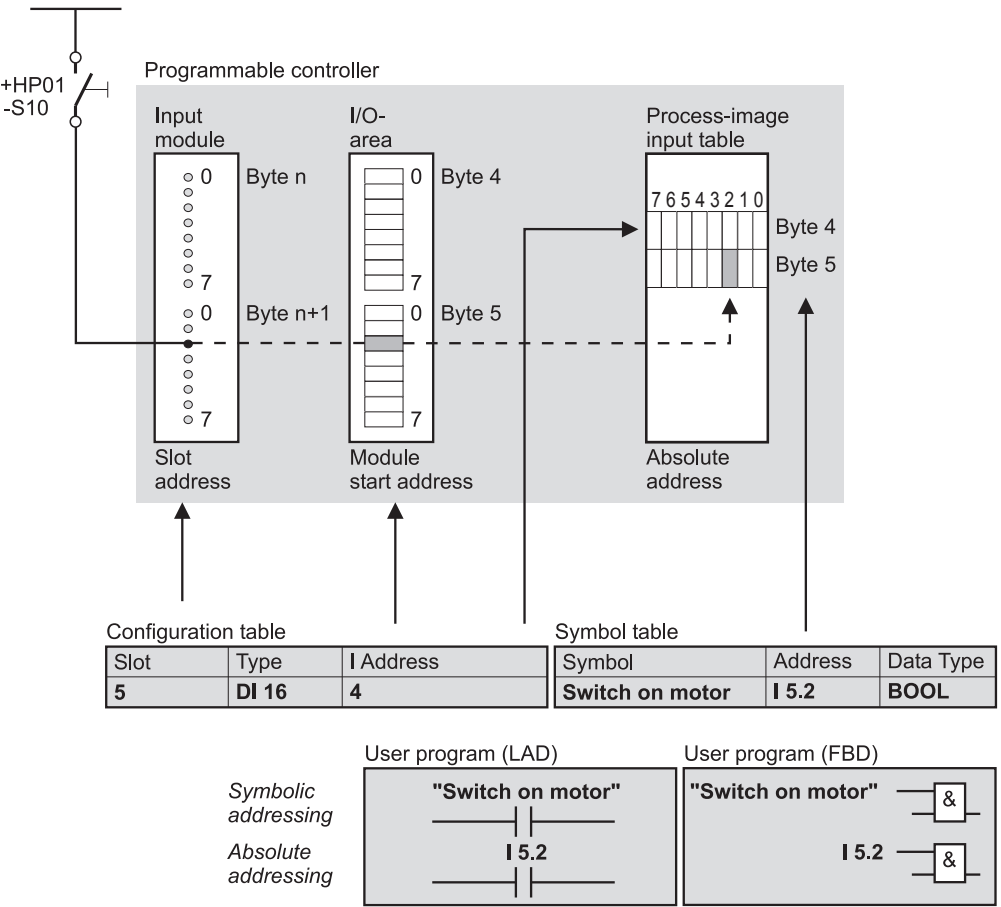


Figure 1.9 Correlation between Module Address, Absolute Address and Symbolic Address (Path of a Signal from Sensor to Scanning in the Program)

If the module contains interface cards, each of these cards is also assigned a submodule address. In this way, each binary and analog signal and each serial connection in the system has its own unique address.

Correspondingly, distributed I/O modules also have a “geographical address”. In this case, the number of the DP master system or the PROFINET IO system and the station number replace the rack number.

You use STEP 7's “Hardware Configuration” tool to plan the hardware configuration of an S7 station as per the physical location of the modules. This tool also makes it possible to set the module start addresses and parameterize the modules (see Chapter 2.3 “Configuring Stations”).

1.4.3 Logical Address

The logical address corresponds to the absolute address. It is also referred to as the user data address, for you use this address to access the user data of the input/output modules in the user program, either via the process image (inputs I and outputs Q) or directly on the modules (peripheral inputs PI and peripheral outputs PQ). The range of logical addresses commences at zero and ends at an upper limit specific to the CPU.

With digital modules, the individual signals (the individual bits) are combined into bunches of 8, referred to as bytes. Modules exist with one, two or four bytes. These bytes have the relative addresses 0, 1, 2 and 3; addressing of the bytes commences at the modules start address. Example: with a digital module with four bytes and the start address 8, the individual bytes with addresses 8, 9, 10 and 11 are addressed. With analog modules, the individual analog signals (voltages, currents) are referred to as “channels”, each of which requires two bytes. Depending on their design, analog modules exist with 2, 4, 8 or 16 channels corresponding to an address range of 4, 8, 16 or 32 bytes.

Using the hardware configuration, you assign a logical addresses to each byte of a used module. The addresses are assigned as standard starting at zero, but you can change the proposed address. The logical addresses of the individual modules must not overlap. The logical

addresses are defined separately for the input and output modules, so that an input byte can have the same number as an output byte.

The user data of the distributed I/O are also addressed in bytes using a logical address. In order to guarantee an unambiguous assignment of all user data of a CPU (or more exactly: all user data on a P bus), the logical addresses of the distributed I/O must not overlap with the logical addresses of the central modules.

It is usually the case that the digital modules are assigned according to addresses in the process image so that their signal statuses are automatically updated and they can be addressed using the address areas “Input” and “Output”. Analog modules, FM modules and CP modules are assigned an address which is not within the process image.

1.4.4 Module Start Address

The module start address is the smallest logical (user data) addresses of a module; it identifies the relative byte zero of the module. The following module bytes are then assigned successively with addresses.

In the case of mixed modules containing input and output ranges, the lower range start address is defined as the module start address. If the input and output ranges have the same start address, use the input address.

Using the hardware configuration, you determine the position of the user data addresses in the address volume of the CPU by defining the module start addresses. The lowest logical address is also the module start address for the modules of the distributed I/O and even for the virtual slots in the transfer memory of an intelligent DP slave.

The modules start address is used in many cases to identify a module. Other than this, it has no special meaning.

1.4.5 Diagnostics Address

Appropriately equipped modules can supply diagnostics data that you can evaluate in your program. If centralized modules have a user data address (module start address), you access the module via this address when reading the

diagnostics data. If the modules have no user data address (e.g. power supplies), or if they are part of the distributed I/O, there is a diagnostics address for this purpose.

The diagnostics address is always an address in the I/O input area and occupies one byte. The user data length of this address is zero; if it is located in the process image, as is permitted, it is not taken into account by the CPU when updating the process image.

STEP 7 automatically assigns the diagnostics address counting down from the highest possible I/O address. You can change the diagnostics address with the Hardware Configuration function.

The diagnostics data can only be read with special system functions; accessing this address with load statements has no effect (see also Chapter 20.4 “Communication via Distributed I/O”).

1.4.6 Addresses for Bus Nodes

MPI

Modules that are nodes on an MPI network (CPUs, FMs and CPs) also have an **MPI address**. This address is decisive for the link to programming devices, human machine interface devices and for global data communications.

Please note that with older revision levels of the S7-300 CPUs, the FM and CP modules operated in the same station receive an MPI address derived from the MPI address of the CPU. In the case of newer S7-300 CPUs, the MPI addresses of FM and CP modules in the same station can be determined independently of the MPI address of the CPU.

PROFIBUS DP

Each DP station (e.g. DP master, DP slave, programming device) on the PROFIBUS also has a **node address** (station number) with which it can be unambiguously addressed on the bus.

PROFINET IO

Stations on the Industrial Ethernet have a factory-set **MAC address** which is unique worldwide. An **IP address** is additionally required for identification on the bus, and is configured

for the IO controller. The IP addresses for the IO devices are derived from the IP address of the IO controller. The IO controller (the interface) and each IO device is additionally assigned a **device name**. The IO device is addressed by the user program by means of a **device number** (station number).

1.5 Address Areas

The address areas available in every programmable controller are

- ▷ the peripheral inputs and outputs
- ▷ the process input image and the process output image
- ▷ the bit memory area
- ▷ the timer and counter functions (see Chapters 7 “Timers” and 8 “Counters”)
- ▷ the L stack (see Chapter 18.1.5 “Temporary Local Data”)

To this are added the code and data blocks with the block-local variables, depending on the user program.

1.5.1 User Data Area

In SIMATIC S7, each module can have two address areas: a user data area, which can be directly addressed with Load and Transfer statements, and a system data area for transferring data records.

When modules are accessed, it makes no difference whether they are in racks with centralized configuration or used as distributed I/O. All modules occupy the same (logical) address space.

A module's user data properties depend on the module type. In the case of signal modules, they are either digital or analog input/output signals, and in the case of function modules and communications processors, they might, for example, be control or status information. The volume of user data is module-specific. There are modules that occupy one, two, four or more bytes in this area. Addressing always begins at relative byte 0. The address of byte 0 is the

module start address; it is stipulated in the configuration table.

The user data represent the I/O address area, divided, depending on the direction of transfer, into peripheral inputs (PIs) and peripheral outputs (PQs). If the user data are in the area of the process images, the CPU automatically handles the transfers when updating the process images.

Peripheral inputs

You use the peripheral input (PI) address area when you read from the user data area on input modules. Part of the PI address area leads to the process image. This part always begins at I/O address 0; the length of the area is CPU-specific.

With a Direct I/O Read operation, you can access the modules whose interfaces do not lead to the process input image (for instance analog input modules). The signal states of modules that lead to the process input image can also be read with a Direct Read operation. The momentary signal states of the input bits are then scanned. Please note that this signal state may differ from the relevant inputs in the process image since the process input image is updated at the beginning of the program scan.

Peripheral inputs may occupy the same absolute addresses as peripheral outputs.

Peripheral outputs

You use the peripheral output (PQ) address area when you write values to the user data area on an output module. Part of the PQ address area leads to the process image. This part always begins at I/O address 0; the length of the area is CPU-specific.

With a Direct I/O Write operation, you can access modules whose interfaces do not lead to the process output image (such as analog output modules). The signal states of modules controlled by the process output image can also be directly affected. The signal states of the output bits then change immediately. Please note that a Direct I/O Write operation also updates the signal states of the relevant modules in the process output image! Thus, there is no difference between the process output image and the signal states on the output modules.

Peripheral outputs can reserve the same absolute addresses as peripheral inputs.

1.5.2 Process Image

The process image contains the image of the digital input and digital output modules, and is thus divided into process input image and process output image. The process input image is accessed via the address area for inputs (I), the process output image via the address area for outputs (Q). As a rule, the machine or process is controlled via the inputs and outputs.

The process image can be divided into subsidiary process images that can be updated either automatically or via the user program. Please refer to Chapter 20.2.1 “Process Image Updating” for more details.

On the S7-300 CPUs and, from 10/98, also on S7-400 CPUs, you can use the addresses of the process image not occupied by modules as additional memory area similar to the bit memory area. This applies both for the process input image and the process output image.

On suitably equipped CPUs, say, the CPU 417, the size of the process image can be parameterized. If you enlarge the process image, you reduce the size of the work memory accordingly. Following a change to the size of the process image, the CPU executes initialization of the work memory, with the same effect as a cold restart.

Inputs

An input is an image of the corresponding bit on a digital input module. Scanning an input is the same as scanning the bit on the module itself. Prior to program execution in every program cycle, the CPU's operating system copies the signal state from the module to the process input image.

The use of a process input image has many advantages:

- ▷ Inputs can be scanned and linked bit by bit (I/O bits cannot be directly addressed).
- ▷ Scanning an input is much faster than accessing an input module (for example, you avoid the transient recovery time on the I/O bus, and the system memory response

times are shorter than the module's response times). The program is therefore executed that much more quickly.

- ▷ The signal state of an input is the same throughout the entire program cycle (there is data consistency throughout a program cycle). When a bit on an input module changes, the change in the signal state is transferred to the input at the start of the next program cycle.
- ▷ Inputs can also be set and reset because they are located in random access memory. Digital input modules can only be read. Inputs can be set during debugging or startup to simulate sensor states, thus simplifying program testing.

These advantages are offset by an increased program response time (please also refer to Chapter 20.2.4 “Response Time”).

Outputs

An output is an image of the corresponding bit on a digital output module. Setting an output is the same as setting the bit on the output module itself. The CPU's operating system copies the signal state from the process output image to the module.

The use of a process output image has many advantages:

- ▷ Outputs can be set and reset bit by bit (direct addressing of I/O bits is not possible).
- ▷ Setting an output is much faster than accessing an output module (for example, you avoid the transient recovery time on the I/O bus, and the system memory response times are shorter than the module response times). The program is therefore executed that much more quickly.
- ▷ A multiple signal state change at an output during a program cycle does not affect the bit on the output module. It is the signal state of the output at the end of the program cycle that is transferred to the module.
- ▷ Outputs can also be scanned because they are located in random access memory. While it is possible to write to digital output modules, it is not possible to read them. The scanning and linking of the outputs makes

additional storage of the output bit to be scanned unnecessary.

These advantages are offset by an increased program response time. Chapter 20.2.4 “Response Time” describes how a programmable controller's response time comes about.

1.5.3 Consistent User Data

Data consistency means that data can be handled in a block. Transfer of a data block must not be interrupted, and it is not permissible for the data source or target to be changed from the other end during a transmission either. For example, if you transfer four bytes individually, the transmitting program can be interrupted by a program of higher priority between each byte, and this program could change the data in the source or target area.

In the case of direct access to user data (loading and transferring), the data are read and written as byte, word or doubleword. The load and transfer instructions, upon which the MOVE box with LAD/FBD and the assignment of variables with elementary data types with SCL are based, are designed as interruptible. If you wish to transfer a data block with more than four bytes without interruption between system memory and work memory, use the system function SFC 81 UBLKMOV.

Data transfer between a DP slave and DP master is consistent for a complete slave even if e.g. the transfer area for an intelligent DP slave is divided into several consistent blocks. Data consistency with internode communication is the same as with direct access (1-, 2- and 4-byte consistency). This similarly applies to data transfer between IO controller and IO devices on the PROFINET IO.

When configuring stations of the distributed I/O with three or more than four bytes of user data, you can specify the consistent user data areas. These areas are transferred consistent to the parameterized target area (e.g. data area in work memory or process image) using the system functions SFC 14 DPRD_DAT and SFC 15 DPWR_DAT.

Please note that the “normal” updating of process images can be interrupted following each transmitted doubleword. An exception with

newer CPUs is the transfer of user data blocks for distributed I/O using a partial process image if the user data blocks can be configured as consistent using the hardware configuration. You can also influence these data blocks in the process image using a direct access, but you could also possibly destroy the data consistency.

CPU-specific data apply to the maximum size of a consistent area for data transfer with global data communications, S7 basic communications and S7 communications through the operating system (see Technical specifications in the CPU manual).

Diagnostics data and parameters are always transferred consistently in data records (e.g. diagnostics data with the SFC 13 DPMRM_DG or SFB 54 RALRM, or parameter data transferred to and from modules with the SFB 52 RDREC and SFB 53 WRREC).

1.5.4 Bit Memories

The area called bit memories holds what could be regarded as the controller's "auxiliary contactors". Bit memories are used primarily for storing binary signal states. The bits in this area can be treated as outputs, but are not "externalized". Bit memories are located in the CPU's system memory area, and is therefore available at all times. The number of bits in bit memories is CPU-specific.

Bit memories are used to store intermediate results that are valid beyond block boundaries and are processed in more than one block. Besides the data in global data blocks, the following are also available for storing intermediate results

- ▷ Temporary local data, which are available in all blocks but valid for the current block call only, and
- ▷ Static local data, which are available only in function blocks but valid over multiple block calls.

Retentive bit memories

Part of bit memories may be designated "retentive", which means that the bits in that part of bit memories retain their signal states even under off-circuit conditions. Retentivity always begins with memory byte 0 and ends at the designated location. Retentivity is set when the CPU is parameterized. Please refer to Chapter 22.2.4 "Retentivity" for additional information.

Clock memories

Many procedures in the controller require a periodic signal. Such a signal can be implemented using timers (clock pulse generator), watchdog interrupts (time-controlled program execution), or simply by using clock memories.

Clock memories consist of bits whose signal states change periodically with a mark-to-space ratio of 1:1. The bits are combined into a byte, and correspond to fixed frequencies (Figure 1.10). You specify the number of clock memory bits when you parameterize the CPU. Please note that the updating of clock memories is asynchronous to execution of the main program.

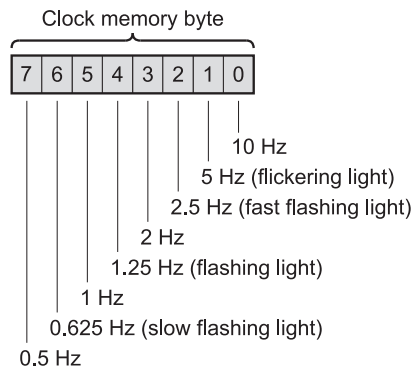


Figure 1.10 Contents of the Clock Memory Byte

2 STEP 7 Programming Software

2.1 STEP 7 Basis Package

This chapter describes the STEP 7 basic package, Version 5.5. While the first chapter presented an overview of the properties of the programmable controller, this chapter tells you how to set these properties.

The basic package contains the statement list (STL), ladder logic (LAD) and function block diagram (FBD) programming languages. In addition to the basic package, option packages such as S7-SCL (Structured Control Language), S7-GRAPH (sequence planning) and S7-HiGraph (state-transition diagram) are also available.

2.1.1 Installation

STEP 7 V5.5 is a 32-bit application which executes with MS Windows XP Professional and MS Windows 7 Professional, Ultimate and Enterprise. MS Internet Explorer V6.0 or higher is required under all operating systems. You require administrator privileges in order to install STEP 7, and you must be registered at least as a main user in order to work with STEP 7.

If you wish to work rapidly with STEP 7 or process large projects, e.g. with several hundred modules, you should use a programming device or PC with up-to-date processing power.

STEP 7 V5.5 occupies approximately 650 to 900 MB on the hard disk depending on the scope of installation and the number of installed languages. A swap-out file is also needed, whose size must be at least twice that of the main memory.

You should ensure there is sufficient memory on the drive containing your project data. The

memory requirements may increase for certain operations, such as copying a project. If there is insufficient space for the swap-out file, errors such as program crashes may occur. You are recommended not to store the project data on the drive containing the Windows swap-out file.

The SETUP program on the DVD is used for the installation, or STEP 7 is already factory-installed on the programming device. In addition to STEP 7, the DVD also includes, inter alia, the Automation License Manager (see Chapter 2.1.2 “Automation License Manager”) and the STEP 7 electronic manuals with Acrobat Reader.

A bus interface is required for the online connection to a programmable controller. This can be a multi-point interface, a PROFIBUS interface, or an Ethernet interface.

An MPI interface is needed for the online connection to a programmable controller. The programming devices have the multipoint interface already built in, but PCs must be retrofitted with an MPI module. If you want to use PC memory cards or micro memory cards, you will need a prommer.

STEP 7 V5 has multi-user capability, that is, a project that is stored, say, on a central server can be edited simultaneously from several workstations. You make the necessary settings in the Windows Control Panel with the “SIMATIC Workstation” program. In the dialog box that appears, you can parameterize the workstation as a single-user system or a multi-user system with the protocols used.

Deinstallation of STEP 7 is carried out with the setup program or in the usual manner for MS Windows using the “Software” program in the Windows Control Panel.

2.1.2 Automation License Manager

A license (right of use) is required to operate STEP 7. This consists of the certificate of license and the electronic license key. The license key is provided on the license key disk or on a USB flash drive.

A license key can be present on the license key disk, on a USB stick and on local or networked hard disks. A license key will only function if it is present on a hard disk with write access. You use the *Automation License Manager* to transfer and administer the license keys. Installation of the Automation License Manager is a requirement for operating STEP 7. You can install the Automation License Manager together with STEP 7 or on its own.

The type of license key is defined in the certificate of license:

- ▷ **Single License**
This license is applicable for an unlimited time, and permissible on any one computer.
- ▷ **Floating License**
This license is applicable for an unlimited time, and provided for procurement via a network.
- ▷ **Trial License**
This license is limited to 14 days, or to a certain number of days starting with its initial use. It can be used for testing and validation.
- ▷ **Upgrade License**
This license permits upgrading of an authorization/license key from a previous version to the current version.

During installation of STEP 7, licensing will be requested if an appropriate license key is not yet present on the hard disk. You can also carry out licensing at a later point in time.

The license key is saved on the hard disk in specially identified blocks. To prevent unintentional destruction of the license key, please observe the information on the handling of license keys provided in the help function of the Automation License Manager.

2.1.3 SIMATIC Manager

The SIMATIC Manager is the main tool in STEP 7; you will find its icon in Windows.



The SIMATIC Manager is started by double-clicking on its icon.

When first started, the project wizard is displayed. This can be used for simple creation of new projects. You can deactivate it with the checkbox "Display Wizard on starting the SIMATIC Manager" and the "Cancel" button, since it can also be called, if required, via the menu command FILE → Wizard 'New Project'.

Programming begins with opening or creating a "project". The example projects supplied are a good basis for familiarization.

When you open example project ZEn01_09_STEP7_Zebra with FILE → OPEN, you will see the split project window: on the left is the structure of the open object (the object hierarchy), and on the right is the selected object. Clicking on the box containing a plus sign in the left window displays additional levels of the structure; selecting an object in the left half of the window displays its contents in the right half of the window (Figure 2.1).

Under the SIMATIC Manager, you work with the objects in the STEP 7 world. These "logical" objects correspond to "real" objects in your plant. A project contains the entire plant, a station corresponds to a programmable controller. A project may contain several stations connected to one another, for example, via an MPI subnet. A station contains a CPU, and the CPU contains a program, in our case an S7 program. This program, in turn, is a "container" for other objects, such as the object *Blocks*, which contains, among other things, the compiled blocks.

The STEP 7 objects are connected to one another via a tree structure. Figure 2.2 shows the most important parts of the tree structure (the "main branch", as it were) when you are working with the STEP 7 basic package for S7

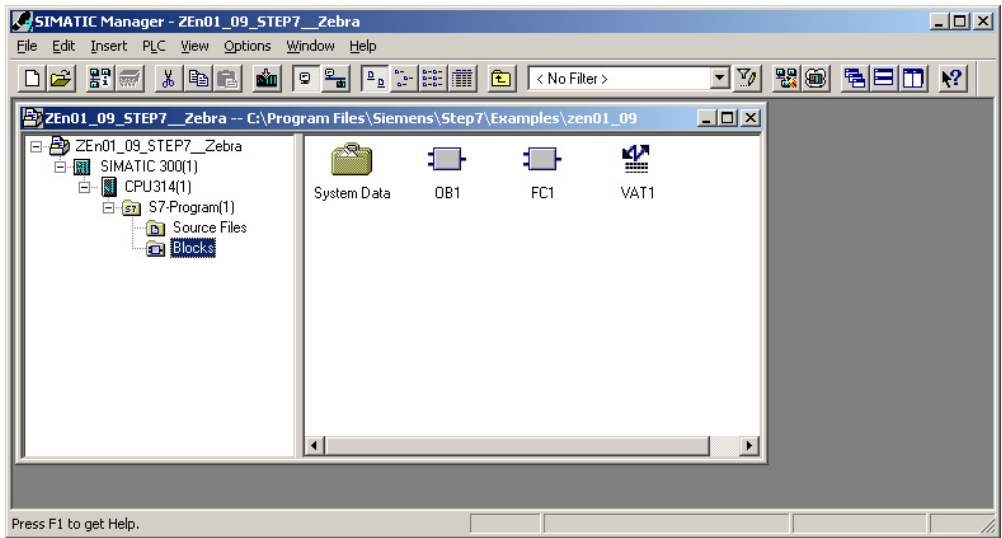


Figure 2.1 SIMATIC Manager Example

applications in offline view. The objects shown in bold type are containers for other objects.

All objects in the Figure are available to you in the offline view. These are the objects that are on the programming device's hard disk. If your programming device is online on a CPU (normally a PLC target system), you can switch to the online view by selecting **VIEW → ONLINE**. This option displays yet another project window containing the objects on the destination device; the objects shown in italics in the Figure are then no longer included.

You can see from the title bar of the active project window whether you are working offline or online. For clearer differentiation, the title bar and the window title can be set to a different color than the offline window. For this purpose, select **OPTIONS → CUSTOMIZE** and modify the entries in the “View” tab.

Select **OPTIONS → CUSTOMIZE** to change the SIMATIC Manager's basic settings, such as the session language, the archive program and the storage location for projects and libraries, and configuring the archive program.

Editing sequences

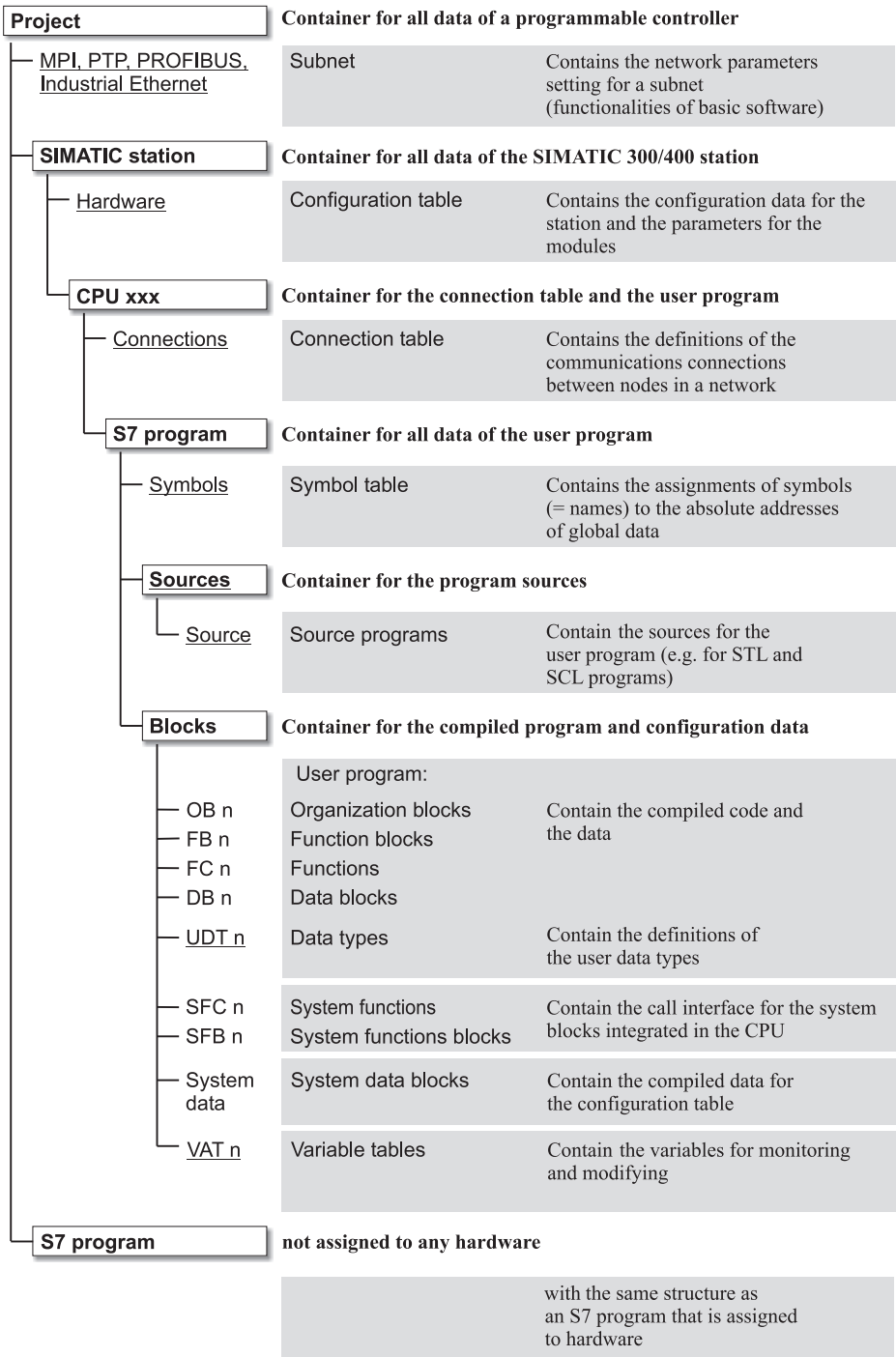
The following applies for the general editing of objects:

To *select an object* means to click on it once with the mouse so that it is highlighted (this is possible in both halves of the project window).

To *name an object* means to click on the name of the selected object (a frame will appear around the name and you can change the name in the window) or select the menu item **EDIT → OBJECT PROPERTIES** and change the name in the dialog box. With some objects such as a CPU, you can only change the name with the relevant tool (application), in this case with the Hardware Configuration.

To *open an object*, double-click on that object. If the object is a container for other objects, the SIMATIC Manager displays the contents of the object in the right half of the window. If the object is on the lowest hierarchical level, the SIMATIC Manager starts the appropriate tool for editing the object (for instance, double-clicking on a block starts the editor, allowing the block to be edited).

In this book, the menu items in the standard menu bar at the top of the window are described



(The underlined objects are only present in the offline data management.)

Figure 2.2 Object Hierarchy in a STEP 7 Project

as *operator sequences*. Programmers experienced in the use of the operator interface use the icons from the toolbar. The use of the *right mouse button* is very effective. Clicking on an object once with the right mouse button screens a menu showing the current editing options.

2.1.4 Projects and Libraries

In STEP 7, the “main objects” at the top of the object hierarchy are projects and libraries. Starting with STEP 7 V5.2, you can combine projects and libraries into multiprojects (see Chapter 2.1.5 “Multiprojects”).

Projects are used for the systematic storing of data and programs needed for solving an automation task. Essentially, these are

- ▷ the hardware configuration data,
- ▷ the parameterization data for the modules,
- ▷ the configuring data for communication via networks,
- ▷ the programs (code and data, symbols, sources).

The objects in a project are arranged hierarchically. The opening of a project is the first step in editing all (subordinate) objects which that object contains. The following sections discuss how to edit these objects.

Libraries are used for storing reusable program components. Libraries are organized hierarchically. They may contain STEP 7 programs which in turn may contain a user program (a container for compiled blocks), a container for source programs, and a symbol table. With the exception of online connections (no debugging possible), the creation of a program or program section in a library provides the same functionality as in an object.

As supplied, STEP 7 V5 provides the *Standard Library* containing the following programs:

- ▷ System Function Blocks
Contains the call interfaces of the system blocks for offline programming integrated in the CPU
- ▷ S5-S7 Converting Blocks
Contains loadable functions for the S5/S7 converter (replacement of S5 standard func-

tion blocks in conjunction with program conversion)

- ▷ T1-S7 Converting Blocks
Contains additional loadable functions and function blocks for the T1-S7 converter
- ▷ IEC Function Blocks
Contains loadable functions for editing variables of the complex data types DATE_AND_TIME and STRING
- ▷ Communication Blocks
Contains loadable functions for controlling CP modules
- ▷ Miscellaneous Blocks
Contains blocks for time stamping and time synchronization
- ▷ PID Control Blocks
Contains loadable function blocks for closed-loop control
- ▷ Organization Blocks
Contains the templates for the organization blocks (essentially the variable declaration for the start information)

You will find an overview of the contents of these libraries in Chapter 25 “Block Libraries”. Should you, for example, purchase an S7 module with standard blocks, the associated installation program installs the standard blocks as a library on the hard disk. You can then copy these blocks from the library to your project. A library is opened with FILE → OPEN, and can then be edited in the same way as a project. You can also create your own libraries.

The menu item FILE → NEW generates a new object at the top of the object hierarchy (project, library). The location in the directory structure where the SIMATIC Manager is to create a project or library must be specified under the menu item OPTIONS → CUSTOMIZE or in the “New dialog” box.

The INSERT menu is used to add new objects to existing ones (such as adding a new block to a program). Before doing so, however, you must first select the object container in which you want to insert the new object from the left half of the SIMATIC Manager window.

You copy object containers and objects with EDIT → COPY and EDIT → PASTE or, as is usual with Windows, by dragging the selected object

with the mouse from one window and dropping it in another. Please note that you cannot undo deletion of an object or an object container in the SIMATIC Manager.

2.1.5 Multiprojects

In a multiproject, projects and libraries are combined in an entity. The multiproject allows processing of communications connections such as S7 connections between the projects. A multiproject can then be handled almost like a single project. Limitations: stations connected together by means of direct data exchange (“internode communication”) or through global data communication must be present in the same project.

In a multiproject, it is possible to carry out parallel processing of individual projects by various employees without problem. The individual projects can be present in different directories in a networked environment. The cross-project functions, such as the matching of sub-networks and connections, are then carried out centrally when processing the multiproject.

It is also advantageous to create a multiproject if you wish to make the individual projects smaller and clearer.

You can archive and retrieve a multiproject just like a project or library.

2.1.6 Online Help

The SIMATIC Manager's online help provides information you need during your programming session without the need to refer to hard-copy manuals. You can select the topics you need information on by selecting the HELP menu. The online help option GETTING STARTED, for instance, provides a brief summary on how to use the SIMATIC Manager.

HELP → CONTENTS starts the central STEP 7 Help function from any application. This contains all the basic knowledge. If you click on the “Home” symbol in the menu bar (start page), you will be provided with an introduction to the central topics of STEP 7: Starting with STEP 7, Configuring & programming, Tests & troubleshooting, as well as SIMATIC on the Internet.

HELP → CONTEXT-SENSITIVE HELP F1 provides context-sensitive help, i.e. if you press F1, you get information concerning an object selected by the mouse or concerning the current error message.

In the symbol bar, there is a button with an arrow and a question mark. If you click on this button, a question mark is added to the mouse pointer. With this “Help” mouse pointer, you can now click on an object on the screen, e.g. a symbol or a menu command, and you will get the associated online help.

2.2 Editing Projects

When you set up a project, you create “containers” for the resulting data, then you generate the data and fill these containers. Normally, you create a project with the relevant hardware, configure the hardware, or at least the CPU, and receive in return containers for the user program. However, you can also put an S7 program directly into the project container without involving any hardware at all. Note that initializing of the modules (address modifications, CPU settings, configuring connections) is possible only with the Hardware Configuration tool.

We strongly recommend that the entire project editing process be carried out using the SIMATIC Manager. Creating, copying or deleting directories or files as well as changing names (!) with the Windows Explorer within the structure of a project can cause problems with the SIMATIC Manager.

2.2.1 Creating Projects

Project wizard

The STEP 7 Wizard helps you in creating a new project. You specify the CPU used and the wizard creates for you a project with an S7 station and the selected CPU as well as an S7 program container, a source container and a block container with the selected organization blocks. You start the project wizard using FILE → “NEW PROJECT” WIZARD.

Creating a project with the S7 station

If you want to create a project “manually”, this section outlines the necessary actions for you. You will find general information on operator entries for object editing in Chapter 2.1.3 “SIMATIC Manager”.

Creating a new project

Select FILE → NEW, enter a name in the dialog box, change the type and storage location if necessary, and confirm with “OK” or RETURN.

Inserting a new station in the project

Select the project and insert a station with INSERT → STATION → SIMATIC 300 STATION (in this case an S7-300).

Configuring a station

Click on the plus box next to the project in the left half of the project window and select the station; the SIMATIC Manager displays the Hardware object in the right half of the window. Double-clicking on *Hardware* starts the Hardware Configuration tool, with which you edit the configuration tables.

If the module catalog is not on the screen, call it up with VIEW → CATALOG.

You begin configuring by selecting the rail with the mouse, for instance under “SIMATIC 300” and “RACK 300”, “holding” it, dragging it to the free portion in the upper half of the station window, and “letting it go” (drag & drop). You then see a table representing the slots on the rail.

Next, select the required modules from the module catalog and, using the procedure described above, drag and drop them in the appropriate slots. To enable further editing of the project structure, a station requires at least one CPU, for instance the CPU 314 in slot 2. You can add all other modules later. Editing of the hardware configuration is discussed in detail in Chapter 2.3 “Configuring Stations”.

Store and compile the station, then close and return to the SIMATIC Manager. In addition to the hardware configuration, the open station now also shows the CPU.

When it configures the CPU, the SIMATIC Manager also creates an S7 program with all objects. The project structure is now complete.

Viewing the contents of the S7 program

Open the CPU; in the right half of the project window you will see the symbols for the *S7 program* and for the *connection table*.

Open the S7 program; the SIMATIC Manager displays the symbols for the compiled user program (the compiled blocks), the container for the source programs, and the symbol table in the right half of the window.

Open the user program (*Blocks*); the SIMATIC Manager displays the symbols for the compiled configuration data (*System data*) and an empty organization block for the main program (OB 1) in the right half of the window.

Editing user program objects

We have now arrived at the lowest level of the object hierarchy. The first time OB 1 is opened, the window with the object properties is displayed and the editor needed to edit the program in the organization block is opened. You add another empty block for incremental programming by highlighting the object *Blocks* and opening the menu command INSERT → S7 BLOCK → Then select the required block type from the list.

When opened, the *System data* object shows a list of available system data blocks. You receive the compiled configuration data. These system data blocks are edited via the *Hardware* object in the container *Station*. You can transfer *System data* to the CPU with PLC → DOWNLOAD and parameterize the CPU in this way.

The object container *Sources* is empty. With *Sources* selected, you can select INSERT → S7 SOFTWARE → STL SOURCE to insert an empty source text file or you can select INSERT → EXTERNAL SOURCE to transfer a source text file created, say, with another editor in ASCII format to the *Sources* container.

Creating a project without an S7 station

If you wish, you can create a program without first having to configure a station. To do so, generate the container for your program yourself. Select the project and generate an S7 pro-

gram with INSERT → PROGRAM → S7-PROGRAM. Under this S7 program, the SIMATIC Manager creates the object containers *Sources* and *Blocks*. *Blocks* contains an empty OB 1.

Creating a library

You can also create a program under a library, for instance if you want to use it more than once. In this way, the standard program is always available and you can copy it entire or in part into your current program.

Please note that you cannot establish online connections in a library, which means that you can debug a STEP 7 program only within a project.

2.2.2 Managing, Reorganizing and Archiving

The SIMATIC Manager maintains a list of all known “main objects”, arranged according to user projects, libraries, example projects and multiprojects. You install the example projects and the standard libraries in conjunction with STEP 7 and you install the user projects, the multiprojects and your own libraries yourself.

When you execute FILE → MANAGE, the SIMATIC Manager shows you a list of all known projects and libraries with name and path. You can then delete from the list projects or libraries you no longer want to display (“Hide”) or include in the list new projects and libraries (“Display”).

When it executes FILE → REORGANIZE, the SIMATIC Manager eliminates the gaps created by deletions and optimizes data memory similarly to the way a defragmentation program optimizes the data memory on the hard disk. The reorganization can take some time, depending on the data movements involved.

You can also archive a project or library (FILE → ARCHIVE). In this case, the SIMATIC Manager stores the selected object (the project or library directory with all subdirectories and files) in compressed form in an archive file.

Projects and libraries cannot be edited in the archived (compressed) state. You can unpack an archived object with FILE → RETRIEVE and

then you can edit it further. The retrieved objects are automatically accepted into the project or library management system.

You make the settings for archiving and retrieving on the “Archive” tab under OPTIONS → Customize. You select the archiving program from a drop-down list: Arj (*.arj), PKZip 12.4 (*.zip), or WinZip (*.zip).

You then set the options for archiving on this tab, for example the target directory for archiving and retrieving or “Automatically create archive path” (no further inputs are then required when archiving, since the name of the archive file is generated from the project name).

Archiving a project in the CPU

With the appropriately designed CPUs, you can store a project in archived (compressed) form in the load memory of the CPU, that is, on the memory card. In this way, you can save all project data required for full execution of the user program, such as symbols or source files, direct at the machine or plant. If it becomes necessary to modify or supplement the program, you load the locally stored data onto the hard disk, correct the user program, and save the up-to-date project data again to the CPU.

When loading the project data onto a memory card or micro memory card plugged into the CPU, open the project, mark the CPU and select PLC → SAVE TO MEMORY CARD. In the reverse direction, transfer the stored data back to the programming device with PLC → RETRIEVE FROM MEMORY CARD. Please note that when you write to a memory card plugged into the CPU, the entire contents of the load memory are written to the CPU, including the system data and the user programs.

If you want to fetch back the project data stored on the CPU without creating a project on the hard disk, select the relevant CPU with PLC → DISPLAY ACCESSIBLE NODES. If the memory card is plugged into the module receptacle of the programming device, select the memory card with FILE → S7 MEMORY CARD → OPEN before transferring.

2.2.3 Project Versions

Since STEP 7 V5 has become available, there are three different versions of SIMATIC projects. STEP 7 V1 creates version 1 projects, STEP 7 V2 creates version 2 projects, and STEP 7 V3/V4/V5.0 can be used to create and edit both version 2 and version 3 projects. With STEP 7 from version V5.1, you can create and edit V3 projects and V3 libraries.

If you have a version 1 project, you can convert it into a version 2 project with **FILE → OPEN VERSION 1 PROJECT**. The project structure with the programs, the compiled version 1 blocks, the STL source programs, the symbol table and the hardware configuration remain unchanged.

You can create and edit version 2 projects with STEP 7 versions V2, V3, V4 and V5.0 (Figure 2.3). STEP 7 V5.1 works only with version 3 projects.

Up to STEP 7 Version 5.3 you can convert a V1 project to a V2 project with **FILE → OPEN VERSION 1 PROJECT**. With **FILE → OPEN**, you can open a V2 project and convert it to a V3 project. It is not possible to create a V2 project or save a project as a V2 project.

2.2.4 Creating and editing multiprojects

Using **FILE → NEW** you can create a new multiproject in the SIMATIC Manager in which you select “Multiproject” as the type in the dialog box. With the multiproject selected, you can then generate a new project or a new library in the multiproject using **FILE → MULTIPROJECT → CREATE IN MULTIPROJECT**. You can process the newly created project or library as described in the previous chapters. Using **FILE → MULTIPROJECT → INSERT INTO MULTIPROJECT** you

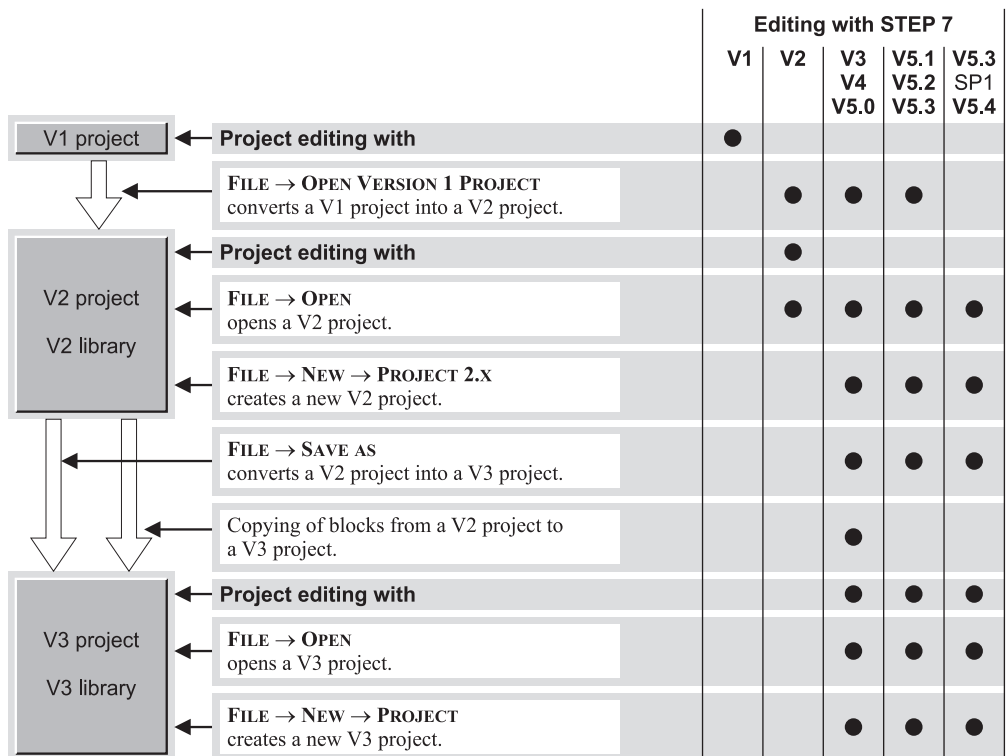


Figure 2.3 Editing Projects with Different Versions

can incorporate existing projects and libraries into the multiproject.

You can also eliminate projects and libraries again from the multiproject: mark the projects: library, and select **FILE → MULTIPROJECT → REMOVE FROM MULTIPROJECT**. The project or library is not deleted in this process.

Using **FILE → MULTIPROJECT → ADJUST PROJECTS** you can start a wizard which supports you in the matching of cross-project connections and when combining subnets (Chapter 2.4.6 “Matching Projects in a Multiproject”).

You can identify one of the libraries in a multiproject as the “master data library” using **FILE → MULTIPROJECT → DEFINE AS MASTER DATA LIBRARY**. This contains, for example, the common blocks of the projects in this multiproject. This library must then only contain one single S7 program.

The menu commands **FILE → SAVE AS**, **FILE → REORGANIZE**, **FILE → MANAGE** and **FILE → ARCHIVE** can also be used on a multiproject, and function as with a single project (see Chapter 2.2.2 “Managing, Reorganizing and Archiving”). In the same manner, archived multiprojects can be transferred to the load memory of a correspondingly designed CPU. There are limitations when archiving a multiproject whose components are distributed among network drives.

2.3 Configuring Stations

You use the Hardware Configuration tool to plan your programmable controller's configuration. Configuring is carried out offline without connection to the CPU. You can also use this tool to address and parameterize the modules. You can create the hardware configuration at the planning stage or you can wait until the hardware has already been installed.

You start the hardware configuration by selecting the station and then **EDIT → OPEN OBJECT** or by double-clicking on the *Hardware* object in the opened container *SIMATIC 300/400 Station*. You make the basic settings of the hardware configuration with **OPTIONS → CUSTOMIZE**.

When configuring has been completed, **STATION → CONSISTENCY CHECK** will show you whether your entries were free of errors. **STATION → SAVE** stores the configuration tables with all parameter assignment data in your project on the hard disk.

STATION → SAVE AND COMPILE not only saves but also compiles the configuration tables and stores the compiled data in the *System data* object in the offline container *Blocks*. After compiling, you can transfer the configuration data to a CPU with **PLC → DOWNLOAD**. The object *System data* in the online container *Blocks* represents the current configuration data on the CPU. You can “return” these data to the hard disk with **PLC → UPLOAD**.

You export the data of the hardware configuration with **STATION → EXPORT**. STEP 7 then creates a file in ASCII format that contains the configuration data and parameterization data of the modules. You can choose between a text format that contains the data in “readable” English characters, or a compact format with hexadecimal data. You can also import a correspondingly structured ASCII file.

Checksum

The Hardware Configuration generates a checksum via a correctly compiled station and stores it in the system data. Identical system configurations have the same checksum so that you can, for example, easily compare an online configuration with an offline configuration.

The checksum is a property of the *System data* object. To read the checksum, open the *Blocks* container in the S7 program, select the *System data* object and open it with **EDIT → OPEN OBJECT**.

The user program also has an appropriate checksum. You can find this along with the checksum of the system data in the properties of *Blocks*: select the *Blocks* container and then **EDIT → OBJECT PROPERTIES** on the “Checksums” tab.

Station window

When opened, the Hardware Configuration displays the station window and the hardware catalog (Figure 2.4). Enlarge or maximize the

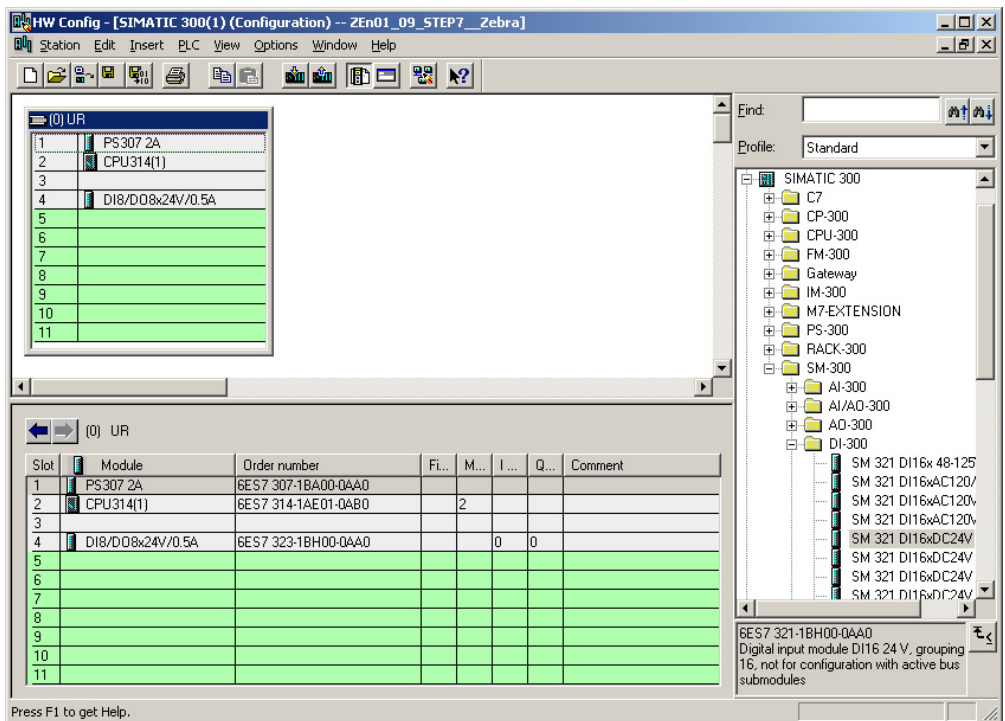


Figure 2.4 Example of a Station Window in the Hardware Configuration

station window to facilitate editing. In the upper section, it displays the S7 stations in the form of tables (one per mounting rack) that are connected together via interface modules when several mounting racks are used. When distributed I/O is connected, the configuration of the DP master system or the PROFINET IO system is specified, with the DP stations and IO devices represented in the form of symbols. The lower section of the station window shows the configuration table that gives a detailed view of the rack or DP slave selected in the upper section.

Hardware catalog

You can fade the hardware catalog in and out with **VIEW → CATALOG**. It contains all available mounting racks, modules and interface submodules known to STEP 7. With **OPTIONS → EDIT CATALOG PROFILE**, you can compile your own hardware catalog that shows only the modules you want to work with – in the struc-

ture you select. By double-clicking on the title bar, you can “dock” the hardware catalog onto the right edge of the station window or release it again.

Installing hardware updates

With **OPTIONS → INSTALL HW UPDATES** you can update components for the hardware catalog. In the following dialogs, select whether you wish to download the update from the Internet or copy it from a CD. Enter the Internet address and the save path. By clicking the “Install” button, the Hardware Configuration transfers the data into the hardware catalog.

Product support information

With **HELP → PRODUCT SUPPORT INFORMATION** you can display information from the Internet for the selected module. You must first enable this function with **OPTIONS → CUSTOMIZE** and set a valid Internet address. The

selected module can be in the hardware catalog or already in the configured rack.

Configuration table

The Hardware Configuration tool works with tables that each represent an S7 station (a mounting rack), a DP station or an IO device. A configuration table shows the slots with the modules arranged in the slots or the properties of the modules such as their addresses and order numbers. A double-click on a module line opens the properties window of the module and allows parameterization of the module.

2.3.1 Arranging Modules

For example, you have created a SIMATIC 300/400 station with the SIMATIC Manager and wish to fit this with an S7 CPU and the associated modules. To do this, open the station (select the station in the SIMATIC Manager and EDIT → Open object or double-click on the *Hardware* object in the open folder *SIMATIC 300/400 STATION*).

You first specify the rack. You can find this for S7-400 stations in the Hardware Catalog under "SIMATIC 400" and "RACK-400" and for S7-300 stations under "SIMATIC 300" and "RACK-300". Select the rack or the DIN rail using the mouse, and drag & drop into any position in the top part of the station window. You are then provided with the empty configuration table for the central rack.

To create a station with an ET 200 CPU, select and open the SIMATIC 300 station in the SIMATIC Manager. In the Hardware Catalog under "PROFIBUS DP" or "PROFINET IO" and "I/O", you can then drag the desired CPU, e.g. IM154-8 CPU under ET 200pro, into the top part of the station window using the mouse or select it using a double-click. The configuration table is then equipped with the CPU.

Next, select the required modules from the module catalog and, in the manner described above, drag and drop them in the appropriate slots. The permissible slots have a green background. A "No Parking" symbol tells you cannot drop the selected module at the intended slot.

Please note for stations with an ET 200 CPU that you may only use the modules which are present under the respective CPU in the Hardware Catalog!

You can also mark the slot to be equipped, and select INSERT → INSERT OBJECT. In a popup window, the Hardware Configuration then shows you all modules permissible for this slot, from which you can select one.

In the case of single-tier S7-300 stations, slot 3 remains empty; it is reserved for the interface module to the expansion rack.

You can generate the configuration table for another rack by dragging the selected rack from the catalog and dropping it in the station window. In S7-400 systems, a non-interconnected rack (or more precisely: the relevant receive interface module) is assigned an interface via the "Link" tab in the Properties window of a Send IM (select module and EDIT → OBJECT PROPERTIES).

The arrangement of distributed I/O stations is described in Chapter 20.4 "Communication via Distributed I/O".

2.3.2 Addressing Modules

When arranging modules, the Hardware Configuration tool automatically assigns a module start address. You can view this address in the lower half of the station window in the object properties for the relevant modules in the "Addresses" tab. If you deselect the option "System default" in this tab for S7-300 modules, you can change the module addresses. When doing so, please observe the addressing rules for S7-300 and S7-400 systems as well as the addressing capacity of the individual modules.

There are modules that have both inputs and outputs for which you can (theoretically) reserve different start addresses. However, please note carefully the special information provided in the product manuals; the large majority of function and communications modules require the same start address for inputs and outputs.

When assigning the module start address, you can also make the assignment to a process image partition in the case of appropriately

designed CPUs. If more than one CPU is inserted in the central rack of an S7-400, multi-computing is automatically set and you must assign the module to a CPU.

With VIEW → ADDRESS OVERVIEW, you get a window containing all the module addresses currently in use for the CPU selected.

Modules on the MPI bus or communications bus have an MPI address. You may also change this address. Note, however, that the new MPI address becomes effective as soon as the configuration data are transferred to the CPU.

Symbols for user data addresses

In the Hardware Configuration tool, you can assign to the inputs and outputs symbols (names) that are transferred to the Symbol Table.

After you have arranged and addressed the digital and analog modules, you save the station data. Then you select the module (line) and EDIT → SYMBOLS. In the window that then opens, you can assign a symbol, a data type and a comment to the absolute address for each channel (bit-by-bit for digital modules and word-by-word for analog modules).

The “Add Symbol” button enters the absolute addresses as symbols in place of the absolute addresses without symbols. The “Apply” button transfers the symbols into the Symbol Table. “OK” also closes the dialog box.

2.3.3 Parameterizing Modules

When you parameterize a module, you define its properties. It is necessary to parameterize a module only when you want to change the default parameters. A requirement for parameterization is that the module is located in a configuration table.

Double-click on the module in the configuration table or select the module and then EDIT → OBJECT PROPERTIES. Several tabs with the specifiable parameters for this module are displayed in the dialog box. When you use this method to parameterize a CPU, you are specifying the run characteristics of your user program.

Some modules allow you to set their parameters at runtime via the user program with the system functions (see Chapter 22.5.2 “System Blocks for Module Parameterization”).

Module identification

Innovated S7 CPUs, PROFIBUS-DPV1 slaves and PROFINET IO devices can support functions for identification and maintenance (I&M functions). For example, you can provide a station with a higher level designation and a location designation and evaluate it later in the program. The higher level designation is used to identify parts of the plant uniquely according to their function. The location designation is part of the item designation and describes, for example, the precise location of a SIMATIC device within a process plant.

To enter the I&M data, select the module in the hardware configuration, then select EDIT → Object Properties, and then – with an appropriately designed module – you can enter the higher level designation and the location designation on the “General” tab or the “Identification” tab. In online mode, you select the module and can then exchange the I&M data between offline data management and the module with PLC → LOAD MODULE IDENTIFICATION or PLC → LOAD MODULE IDENTIFICATION ONTO PG.

To analyze the I&M data, use SFC 51 RDSYST to read the system status list with the system status list ID 16#011C Index 16#0003 for the higher level designation and Index 16#000B for the location designation.

2.3.4 Networking Modules with MPI

You define the nodes for the MPI subsidiary (subnet) with the Module Properties. Select the CPU, or the MPI interface card if the CPU is equipped with one, in the configuration table and open it with EDIT → OBJECT PROPERTIES. The dialog box that then appears contains the “Properties” button in the “Interface” box of the “General” tab. If you click on this button you are taken to another dialog box with a “Parameter” tab where you can find the suitable subnet.

This is also an opportunity to set the MPI address that you have provided for this CPU. Please note that on older S7-300 CPUs, FMs or

CPs with MPI connection automatically receive an MPI address derived from the CPU.

The highest MPI address must be greater than or equal to the highest MPI address assigned in the subnet (take account of automatic assignment of FMs and CPs!). It must have the same value for all nodes in the subnet.

Tip: if you have several stations with the same type of CPUs, assign different names (identifiers) to the CPUs in the different stations. They all have the name “CPUxxx(1)” as default so in the subnet they can only be differentiated by their MPI addresses. If you do not want to assign a name yourself, you can, for example, change the default identifier from “CPUxxx(1)” to “CPUxxx(n)” where “n” is equal to the MPI address.

When assigning the MPI address, please also take into account the possibility of connecting a programming device or operator panel (OP) to the MPI network at a later date for service or maintenance purposes. You should connect permanently installed programming devices or OPs direct to the MPI network; for plug-in devices via a spur line, there is an MPI connector with a heavy-gauge threaded-joint socket. Tip: reserve address 0 for a service programming device, address 1 for a service OP and address 2 for a replacement CPU (corresponds to the default addresses).

2.3.5 Monitoring and Modifying Modules

With the Hardware Configuration, you can carry out a wiring check of the machine or plant without the user program. A requirement for this is that the programming device is connected to a station (online) and the configuration has been saved, compiled and loaded into the CPU. Now you can address every digital and analog module. Select a module and then PLC → MONITOR/MODIFY, and set the Monitor and Modify operating modes and the trigger conditions.

With the “Status Value” button, the Hardware Configuration shows you the signal states or the values of the module channels. The “Modify Value” button writes the value specified in the Modify Value column to the module.

If the “I/O Display” checkbox is active, the peripheral inputs/outputs (module memory) are displayed instead of the inputs/outputs (process image). The “Enable Periph. Outputs” checkbox revokes the output disable of the output modules if the CPU is in STOP mode (see Chapter 2.7.5 “Enabling Peripheral Outputs”).

You can find other methods of monitoring and modifying inputs and outputs in Chapters 2.7.3 “Monitoring and Modifying Variables” and 2.7.4 “Forcing Variables”.

2.4 Configuring the Network

The basis for communications with SIMATIC is the networking of the S7 stations. The required objects are the subnets and the modules with communications capability in the stations. You can create new subnets and stations with the SIMATIC Manager within the project hierarchy. You then add the modules with communications capability (CPUs and CPs) using the Hardware Configuration tool; at the same time, you assign the communications interfaces of these modules to a subnet. You then define the communications relationships between these modules – the connections – with the Network Configuration tool in the connection table.

The Network Configuration tool allows graphical representation and documentation of the configured networks and their nodes. You can also create all necessary subnets and stations with the Network Configuration tool; then you assign the stations to the subnets and parameterize the node properties of the modules with communications capability.

You can proceed as follows to define the communications relationships via the networking configuration tool:

- ▷ Open the MPI subnet created as standard in the project container (if it is no longer available, simply create a new subnet with INSERT → SUBNET).
- ▷ Use the Network Configuration tool to create the necessary stations and – if required – further subnets.
- ▷ Open the stations and provide them with the modules with communications capability.

- Connect the modules with the relevant subnets.
- Adapt the network parameters, if necessary.
- Define the communication connections in the connection table, if required.

You can also configure global data communications within the Network Configuration: select the MPI subnet and then select **OPTIONS** → **DEFINE GLOBAL DATA** (see Chapter 20.5 “Global Data Communication”).

NETWORK → **SAVE** saves an incomplete Network Configuration. You can check the consistency of a Network Configuration with **NETWORK** → **CONSISTENCY CHECK**. You close the Network Configuration with **NETWORK** → **SAVE AND COMPILE**.

Network window

To start the Network Configuration, you must have created a project. Together with the project, the SIMATIC Manager automatically creates an MPI subnet.

A double-click on this or any other subnet starts the Network Configuration. You can also reach the Network Configuration if you open the Connections object in the CPU container.

In the upper section, the Network Configuration window shows all previously created subnets and stations (nodes) in the project with the configured connections (Figure 2.5).

The connection table is displayed in the lower section of the window if a module with “communications capability”, e.g. an S7-400 CPU, is selected in the upper section of the window.

A second window displays the network object catalog with a selection of the available SIMATIC stations, subnets and DP stations. You can fade the catalog in and out with **VIEW** → **CATALOG** and you can “dock” it onto the right edge of the network window (double-click on the title bar). With **VIEW** → **ZOOM IN**, **VIEW** → **ZOOM OUT** and **VIEW** → **ZOOM FACTOR...**, you can adjust the clarity of the graphical representation.

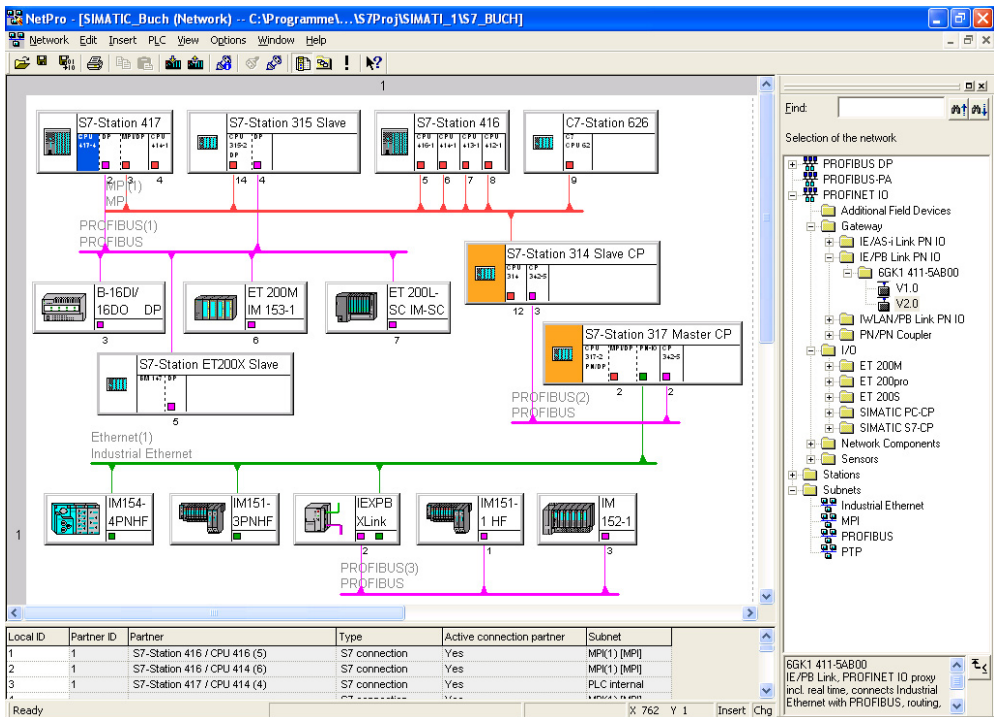


Figure 2.5 Network Configuration Example

2.4.1 Configuring the Network View

Selecting and arranging the components

You begin the Network Configuration by selecting a subnet that you select in the catalog with the mouse, hold and drag to the network window. The subnet is represented in the window as a horizontal line. Impermissible positions are indicated with a “prohibited” sign on the mouse pointer.

You proceed in the same way for the desired stations, at first without connection to the subnet. The stations are still “empty”. A double-click on a station opens the Hardware Configuration tool allowing you to configure the station or at least the module(s) with network connection. Save the station and return to the Network Configuration.

The interface of a module with communications capability is represented in the Network Configuration as a small box under the module view. Click on this box, hold and drag it to the relevant subnet. The connection to the subnet is represented as a vertical line.

Proceed in exactly the same way with all other nodes.

You can move created subnets and stations in the network window. In this way, you can also represent your hardware configuration visually. Under certain circumstances, you get a clearer and more compact arrangement if you reduce represented subnet lengths with VIEW → REDUCED SUBNET LENGTHS.

Setting communications properties

After creating the graphical view, you parameterize the subnets: select the subnets and then EDIT → OBJECT PROPERTIES. The properties window that then appears includes the S7 subnet ID in the “General” tab. The ID consists of two hexadecimal numbers, the project number and the subnet number. You require this S7 subnet ID if you want to go online with the programming device without a suitable project in order to reach other nodes via the subnet. You set the network properties in the “Network Settings” tab, e.g. the data transfer rate or the highest node address.

When you select the network connection of a node, you can define the network properties of the node with EDIT → OBJECT PROPERTIES, e.g., the node address and the subnet it is connected to, or you can create a new subnet.

On the “Interfaces” tab of the station properties, you can see an overview of all modules with communications capability, with the node addresses and the subnet types used.

You define the module properties of the nodes in a similar way (with the same operator inputs as in the Hardware Configuration tool).

2.4.2 Configuring a Distributed I/O with the Network Configuration

You can also use the Network Configuration to configure the distributed I/O with PROFIBUS DP or PROFINET IO. Select VIEW → WITH DP SLAVES/IO DEVICES to display or fade out DP slaves and IO devices in the network view.

PROFIBUS DP

You require the following in order to configure a DP master system:

- ▷ A PROFIBUS subnet (if not already available, drag the PROFIBUS subnet from the network object catalog to the network window),
- ▷ A DP master in a station (if not already available, drag the station from the network object catalog to the network window, open the station and select a DP master with the Hardware Configuration tool, either integrated in the CPU or as an autonomous module),
- ▷ The connection from the DP master to the PROFIBUS subnet (either select the subnet in the Hardware Configuration tool or click on the network connection to the DP master in the Network Configuration, “hold” and drag to the PROFIBUS network).

In the network window, select the DP master to which the slave is to be assigned. Find the DP slave in the network object catalog under “PROFIBUS DP” and the relevant sub-catalog, drag it to the network window and fill out the properties window that appears.

You parameterize the DP slave by selecting it and then selecting EDIT → OPEN OBJECT. The Hardware Configuration is started. Now you can set the user data addresses or, in the case of modular slaves, select the I/O modules (see Chapter 2.3 “Configuring Stations”).

You can only connect an intelligent DP slave to a subnet if you have previously created it (see Chapter 20.4.2 “Configuring PROFIBUS DP”). In the network object catalog, you can find the type of intelligent DP slave under “Already created stations”; drag it, with the DP master selected, to the network window and fill out the properties window that then appears (as in the Hardware Configuration tool).

With VIEW → HIGHLIGHT → MASTER SYSTEM, you emphasize the assignment of the nodes of a DP master system; first, you select the master or a slave of this master system. With VIEW → REARRANGE, the DP slaves are assigned optically to their DP master.

PROFINET IO

In order to configure a PROFINET IO system, you require:

- ▷ An Industrial Ethernet subnet (if not already available, drag the Industrial Ethernet subnet from the network object catalog to the network window)
- ▷ An IO controller in a station (if not already available, drag the station from the network object catalog to the network window, open the station, and select an IO controller with the Hardware Configuration tool, either integrated in the CPU or as an autonomous module)
- ▷ The connection from the IO controller to the Industrial Ethernet subnet (either already select the subnet in the Hardware Configuration tool, or click on the network connection to the IO controller in the Network Configuration, “hold” and drag to the Industrial Ethernet network)

In the network window, select the IO controller to which the IO device is to be assigned. Find the IO device in the network object catalog under “PROFINET IO” and the relevant sub-catalog, drag it to the network window and fill out the properties window that appears.

You parameterize the IO device by selecting it and then selecting EDIT → OPEN OBJECT. The Hardware Configuration is started. Now you can set the user data addresses or the I/O modules (see Chapter 2.3 “Configuring Stations”).

With VIEW → HIGHLIGHT → PROFINET IO SYSTEM, you emphasize the assignment of the nodes of a PROFINET IO system; first, you select the IO controller or an IO device. With VIEW → Rearrange, the IO devices are assigned optically to their IO controller.

2.4.3 Configuring connections

Connections describe the communications relationships between two devices. Connections must be configured if

- ▷ you want to establish S7 communications between two SIMATIC S7 devices (“Communication via configured connections”) or
- ▷ the communications partner is not a SIMATIC S7 device.

Note: you do not require a configured connection for direct online connection of a programming device to the MPI network for programming or debugging. If you want to reach other nodes arranged in other connected subnets with the programming device, you must configure the connection of the programming device: in the Network Object Catalog, select the *PG/PC* object under *Stations* by double-clicking, open *PG/PC* in the network window by double-clicking, and select the interface and assign it to a subnet.

Connection table

The communications connections are configured in the connection table. Requirement: you have created a project with all stations that are to exchange data with each other, and you have assigned the modules with communications capability to a subnet.

The object *Connections* in the *CPU* container represents the connection table. A double-click on *Connections* starts the Network Configuration in the same way as a double-click on a subnet in the project container.

To configure the connections, select e.g. an S7-400 CPU in the Network Configuration. In the

Table 2.1 Connection Table Example

Local ID	Partner ID	Partners	Type	Active connection partner	Send operating mode messages
1	1	Station 416 / CPU416(5)	S7 connection	yes	no
2	2	Station 416 / CPU416(5)	S7 connection	yes	no
3		Station 315 / CPU315(7)	S7 connection	yes	no
4	1	Station 417 / CPU414(4)	S7 connection	yes	no

lower section of the network window, you get the connection table (Table 2.1; if it is not visible, place the mouse pointer on the lower edge of the window until it changes shape and then drag the window edge up). You enter a new communication connection with INSERT → NEW CONNECTION or by double-clicking on an empty line.

You create a connection for each “active” CPU. Please note that you cannot create a connection table for an S7-300 CPU; S7-300 CPUs can only be “passive” partners in an S7 connection.

In the “New Connection” window, you select the communications partner in the “Station” and “Module” dialog boxes (Figure 2.6); the station and the module must already exist. You also determine the connection type in this window.

If you want to set more connection properties, activate the check box “Before inserting: display properties”.

The connection table contains all data of the configured connections. To be able to display this clearly, use VIEW → OPTIMIZE COLUMN WIDTH and VIEW → DISPLAY COLUMNS and select the information you are interested in.

Connection ID

The number of possible connections is CPU-specific. STEP 7 defines a connection ID for every connection and for every partner. You require this specification when you use communications blocks in your program.

You can modify the **local ID** (the connection ID of the currently opened module). This is necessary if you have already programmed communications blocks and you want to use the local ID specified there for the connection.

You enter the new local ID as a hexadecimal number. It must be within the following value ranges, depending on the connection type, and must not already be assigned:

- ▷ Value range for S7 connections:
0001_{hex} to 0FFF_{hex}
- ▷ Value range for S7 connections with loadable S7 communications (S7-300):
0001_{hex} to 008F_{hex}
- ▷ Value range for PtP connections:
1000_{hex} to 1400_{hex}

You change the **partner ID** by going to the connection table of the partner CPU and changing (what is then) the local ID: select the connection line and then EDIT → OBJECT PROPERTIES. If STEP 7 does not enter a partner ID, it is a one-way connection (see below).

Partners

This column displays the connection partner. If you want to reserve a connection resource without naming a partner device, enter “unspecified” in the dialog box under Station.

In a **one-way connection**, communication can only be initiated from one partner; example: S7 communications between an S7-400 and S7-300 CPU. Even without S7 communications functions in the S7-300-CPU, data can be exchanged by an S7-400 CPU with SFB 14 GET and SFB 15 PUT. In the S7-300, no user program runs for this communication but the data exchange is handled by the operating system.

A one-way connection is configured in the connection table of the “active” CPU. Only then does STEP 7 assign a “Local ID”. You also load this connection only in the local station.

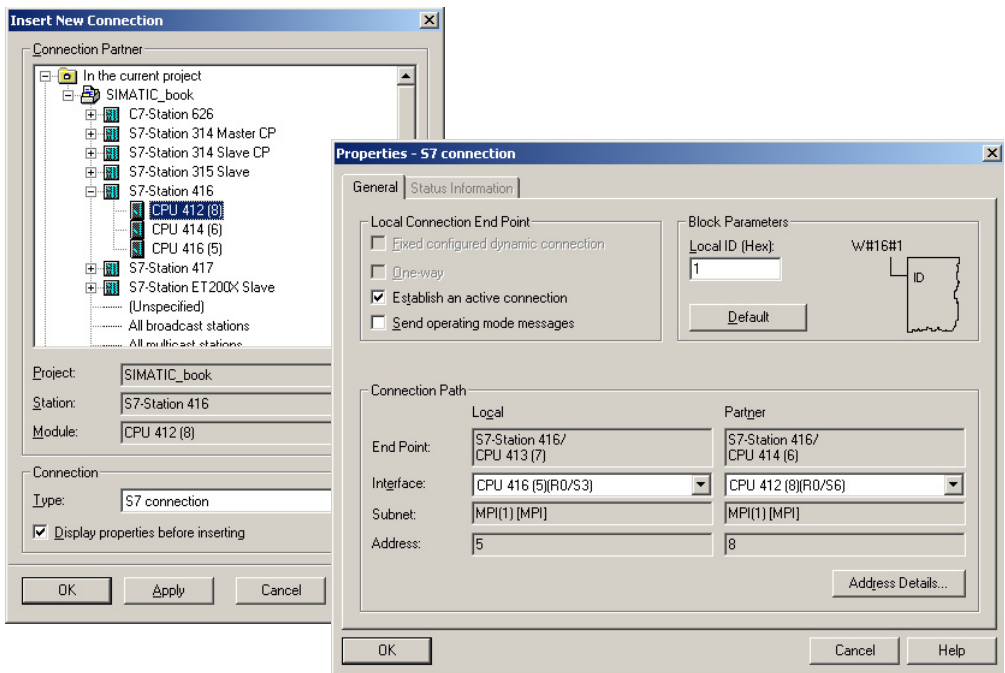


Figure 2.6 Configuring Communications Connections

With a **two-way connection**, both partners can assume communication actively; e.g. two S7-400 CPUs with the communications functions SFB 8 SEND and SFB 9 BRCV.

You configure a two-way connection only once for one of the two partners. STEP 7 then assigns a “Local ID” and a “Partner ID” and generates the connection data for both stations. You must load each partner with its own connection table.

Connection type

The STEP 7 Basic Package provides you with the following connection types in the Network Configuration:

PtP connection, approved for the subnet PTP (3964(R) and RK 512 procedures) with S7 communications. A PtP (point-to-point) connection is a serial connection between two partners. These can be two SIMATIC S7 devices with the relevant interfaces or CPs, or a SIMATIC S7 device and a non-Siemens device, e.g. a printer or a barcode reader.

S7 connection, approved for the subnets MPI, PROFIBUS and Industrial Ethernet with S7 communication. An S7 connection is the connection between SIMATIC S7 devices and can include programming devices and human machine interface devices. Data are exchanged via the S7 connection, or programming and control functions are executed.

Fault-tolerant S7 connection, approved for the subnets PROFIBUS and Industrial Ethernet with S7 communications. A fault-tolerant S7 connection is made between fault-tolerant SIMATIC S7 devices and it can also be established to an appropriately equipped PC.

The software component “SIMATIC NCM”, which is part of STEP 7, is available for **parameterizing CPs**. You have connection types available for selection: FMS connection, FDL connection, ISO transport connection, ISO-on-TCP connection, TCP connection, UDP connection and e-mail connection.

Active connection buildup

Prior to the actual data transfer, the connection must be built up (initialized). If the connection partners have this capability, you specify here which device is to establish the connection. You do this with the check box “Establish an active connection” in the properties window of the connection (select the connection and then EDIT → OBJECT PROPERTIES).

Sending operating state messages

Connection partners with a configured two-way connection can exchange operating state messages. If the local node is to send its operating state messages, activate the relevant check box in the properties window of the connection. In the user program of the partner CPU, these messages can be received with SFB 23 USTATUS.

Connection path

As the connection path, the properties window of the connection displays the end points of the connection and the subnets over which the connection runs. If there are several subnets for selection, STEP 7 selects them in the order Industrial Ethernet before Industrial Ethernet/TCP-IP before MPI before PROFIBUS.

The local station and the partner station with the CPU over which the connection runs are displayed as the end points of the connection.

The modules with communications capability are listed under "Interface", with specification of the interface module, the rack number, and the slot. If both partners can be accessed over several connection paths, the preferred path can be set here. STEP 7 automatically adapts the remaining settings. If both CPUs are located in the same rack (e.g. S7-400 CPUs in multicomputing), the display box shows "PLC-internal".

Under "Subnet" and "Address" you can then see the subnet used and the set node address.

Connections between projects

For data exchange between two S7 modules belonging to different SIMATIC projects, you enter “unspecified” for connection partner in

the connection table (in the local station in both projects).

Please ensure that the connection data agree in both projects (STEP 7 does not check this). After saving and compiling, you load the connection data into the local station in each project.

If a project is to subsequently become part of a multiproject, and if the connection partner is also within a project of the multiproject, select “In unknown project” as the connection partner, and enter an unambiguous connection name (reference) in the properties window.

Connection to non-S7 stations

Within a project, you can also specify stations other than S7 stations as connection partners:

- ▷ Other stations (non-Siemens devices and also S7 stations in another project)
- ▷ Programming devices/PCs
- ▷ SIMATIC S5 stations

A requirement for configuring the connection is that the non-S7 station exists as an object in the project container and you have connected the non-S7 station to the relevant subnet in the station properties (e.g. select the station in the Network Configuration, select EDIT → OBJECT PROPERTIES and connect the station with the desired subnet on the “Interfaces” tab).

2.4.4 Gateways

If the programming device is connected to a subnet, it can reach all other nodes on this subnet. For example, from one connection point, you can program and debug all S7 stations connected to an MPI network. If another subnet such as a PROFIBUS subnet is connected to an S7 station, the programming device can also reach the stations on the other subnet. The requirement for this is that the station with the subnet transition has routing capability, that is, it will channel the transferred message frames.

When the network configuration is compiled, routing tables containing all the necessary information are automatically generated for the stations with subnet transitions. All accessible communications partners must be configured in

a plant network within an S7 project and must be supplied with the “knowledge” of which stations can be reached via which subnets and subnet transitions.

If you want to reach all nodes in a subnet with a programming device from one connection point, you must configure the connection point. You enter a “placeholder”, a PG/PC station from the Network Object Catalog in the network configuration at the relevant subnet. You configure a PG/PC station on every subnet to which you want to connect a programming device.

During operation, you connect the programming device to the subnet and select PLC → ASSIGN PG/PC. This adapts the interfaces of the programming device to the configured settings for the subnet. Before disconnecting the programming device again from the subnet, select PLC → CANCEL PG/PC ASSIGNMENT.

If you go online with a programming device that does not contain the right project, you require the S7 subnet ID for network access. The S7 subnet ID comprises two numbers: the project number and the subnet number. You can obtain the subnet ID in the network configuration by selecting the subnet and then EDIT → OBJECT PROPERTIES on the “General” tab.

2.4.5 Loading the Connection Data

To activate the connections, you must load the connection table into the PLC following saving and compiling (all connection tables into all “active” CPUs).

Requirement: You are in the network window and the connection table is visible. The programming device is a node of the subnet over which the connection data are to be loaded into the modules with communications capability. All subnet nodes have been assigned unique node addresses. The modules to which connection data are to be transferred are in the STOP mode.

With PLC → DOWNLOAD TO CURRENT PROJECT → ..., you transfer the connection and configuration data to the accessible modules. Depending on which object is selected and which menu command is selected, you can choose between the following

- SELECTED STATIONS
- SELECTED AND PARTNER STATIONS
- STATIONS ON THE SUBNET
- SELECTED CONNECTIONS
- CONNECTIONS AND GATEWAYS

In order to delete all connections of a programmable module, load an empty connection table into the associated module.

The compiled connection data are also a component part of the *System data* in the *Blocks* container. Transfer of the system data and the subsequent startup of the CPUs effectively also transfers the connection data to the modules with communications capability.

For online operation via MPI, a programming device requires no additional hardware. If you connect a PC to a network or if you connect a programming device to an Ethernet or PROFIBUS network, you require the relevant interface module. You parameterize the module with the application “Setting the PG/PC Interface” in the Windows Control Panel.

2.4.6 Matching Projects in a Multiproject

When opening a multiproject with the Network Configuration tool, a window is displayed with the projects present in the multiproject. You also obtain this window if you open a project included in a multiproject and select VIEW → MULTIPROJECT. The window displays the projects present in the multiproject and the cross-project subnets which have already been combined. You can now select a project for further processing by double clicking (Figure 2.7).

Projects usually contain communications connections between the individual stations. If projects are combined into a multiproject, or if an existing project is included into the multiproject, these connections can be combined and matched.

If you select VIEW → CROSS-PROJECT NETWORK VIEW in an opened project that belongs to a multiproject, you will see an overview of all stations of the multiproject and the current connections. In the cross-project network view, you cannot make any changes to the projects. Selecting VIEW → CROSS-PROJECT NETWORK VIEW again exits the multiproject view.

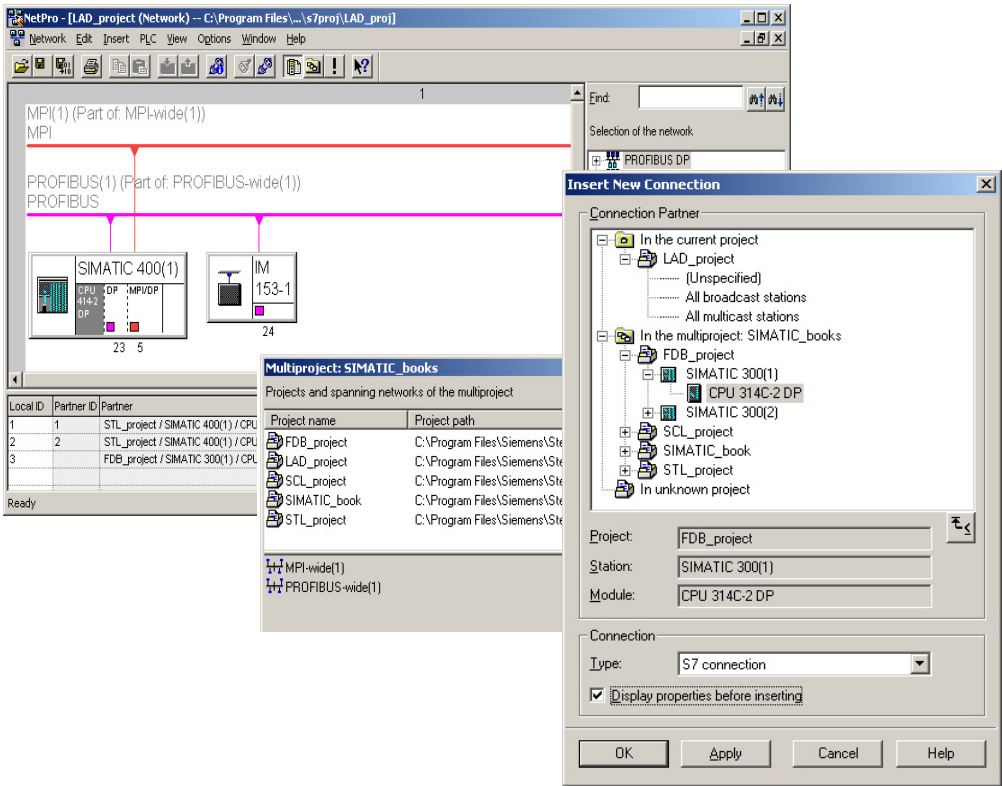


Figure 2.7 Multiproject in the Network Configuration tool

Combining subnets

The MPI, PROFIBUS and Industrial Ethernet subnets are combined together first. A prerequisite is that the subnets to be combined have the same subnet ID. With the subnet selected, you can set these in the Network Configuration tool using EDIT → Object Properties.

With FILE → MULTIPROJECT → ADJUST PROJECTS you can call a wizard for an open multiproject in the SIMATIC Manager which supports you when matching. In the Network Configuration tool, you obtain the dialog window with EDIT → Merge/unmerge subnetworks → ...

You select the type of subnet, click the “Execute” button, and obtain the subnets of the selected type present in the multiproject. You can now select individual subnets of the projects, and combine in a cross-project subnet. You can use

the same dialog to eliminate subnets from the cross-project subnet.

Several cross-project subnets of the same type can be created in a multiproject. The properties of the cross-project subnet are determined by the first subnet added or by the subnet selected with the “Select” button. Use “OK” or “Apply” to acknowledge the settings. Subnets which are part of a cross-project subnet are identified by a different symbol in the SIMATIC Manager.

Combining connections

The connections configured in single projects which lead to a partner in another project can be combined in a multiproject. If you select the partner “In unknown project” when configuring connections in a single project in the window “Insert new connection”, you can subsequently enter a connection name (reference) in

the window “Properties - S7 connection”. Connections in different projects with the same connection names can be combined automatically.

In the SIMATIC Manager, this is carried out by the wizard for project matching if you click “Combine connections” and “Execute”. Connections are then combined which have identical connection names (reference).

In the Network Configuration tool, you can also combine connections with “unspecified” partners. Select EDIT → MERGE CONNECTIONS to obtain a dialog box with all configured connections. Select one connection in each of the Windows “Connections without connection partner” and “Possible connection partners” and click “Assign”. The assigned connections are listed in the bottom window “Assigned connections”. Use “Merge” to then combine the connections. The connections are assigned the properties of the local module of the currently opened project. You can modify the connection properties when combining.

Configuring cross-project connections

Following the combination of subnets, cross-project connections can be configured. The procedure is the same as for project-internal connections, extended by specification of the project at the connection partner.

You can test a network configuration in the multiproject for contradictions with NETWORK → CHECK CROSS-PROJECT CONSISTENCY.

2.5 Creating the S7 Program

2.5.1 Introduction

The user program is created under the object *S7 Program*. You can assign this object in the project hierarchy of a CPU or you can create it independently of a CPU. It contains the object *Symbols* and the containers *Sources* and *Blocks* (Figure 2.8).

With **incremental** program creation, you enter the program direct block-by-block. Entries are checked immediately for syntax. At the same time, the block is compiled as it is saved and

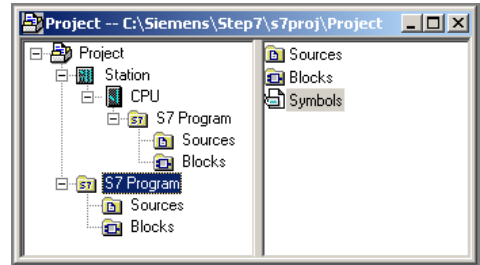


Figure 2.8 Objects for Programming

then stored in the container *Blocks*. With incremental programming, you can also edit blocks online in the CPU, even during operation. Incremental programming is possible in all the basic languages.

In the case of **source-oriented** program creation, you write one or more program sources and store these in the container *Sources*. Program sources are ASCII text files that contain the program statements for one or more blocks, possibly even for the entire program. You compile these sources and you get the compiled blocks in the container *Blocks*. Source-oriented program creation is used in STL and SCL; you cannot use source-oriented programming with LAD or FBD, but programs created with LAD or FBD can be stored as source files.

The signal states or the values of addresses are processed in the program. An address is, for example, the input I1.0 (*absolute addressing*). With the help of the **Symbol Table** under the object *Symbols*, you can assign a symbol (an alphanumeric name, e.g. “Switch motor on”) to an address and then access it with this name (*symbolic addressing*). In the properties of the offline object container *Blocks*, you specify whether in the event of a change in the Symbol Table the absolute address or the symbol is to be definitive for the already compiled blocks when next saved (*address priority*).

2.5.2 Symbol Table

In the control program, you work with addresses; these are inputs, outputs, timers, blocks. You can assign absolute addresses (e.g. I1.0) or symbolic addresses (e.g. Start signal). Symbolic addressing uses names instead of the

absolute address. You can make your program easier to read by using meaningful names.

In symbolic addressing, a distinction is made between *local* symbols and *global* symbols. A local symbol is known only in the block in which it has been defined. You can use the same local symbols in different blocks for different purposes. A global symbol is known throughout the entire program and has the same meaning in all blocks. You define global symbols in the symbol table (object *Symbols* in the container *S7 Program*).

A global symbol starts with an alpha character and can be up to 24 characters long. A global symbol can also contain spaces, special characters and national characters such as the umlaut. Exceptions to this are the characters 00_{hex}, FF_{hex} and the inverted commas ("). You must enclose symbols with special characters in inverted commas when programming. In the compiled block, the program editor displays all global symbols in inverted commas. The symbol comment can be up to 80 characters long.

In the symbol table you can assign names to the following addresses and objects:

- ▷ Inputs I, outputs Q, peripheral inputs PI and peripheral outputs PQ
- ▷ Memory bits M, timer functions T and counter functions C
- ▷ Code blocks OBs, FBs, FCs, SFCs, SFBs and data blocks DBs
- ▷ User data types UDTs
- ▷ Variable table VAT

Data addresses in the data blocks are included among the local addresses; the associated symbols are defined in the declaration section of the data block in the case of global data blocks and in the declaration section of the function block in the case of instance data blocks.

When creating an S7 program, the SIMATIC Manager also creates an empty symbol table *Symbols*. You open this and can then define the global symbols and assign them to absolute addresses (Figure 2.9). There can be only one single symbol table in an S7 program.

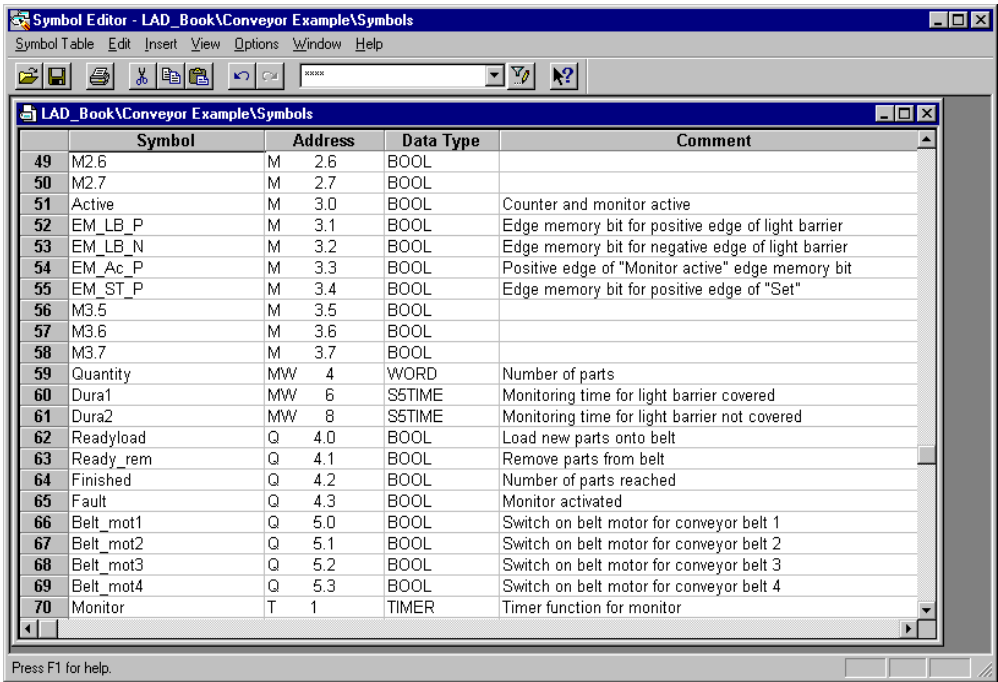


Figure 2.9 Symbol Table Example

The data type is part of the definition of a symbol. It defines specific properties of the data behind the symbol, essentially the representation of the data contents. For example, the data type `BOOL` identifies a binary variable and the data type `INT` designates a digital variable whose contents represent a 16-bit integer. Please refer to Chapter 3.5 “Variables, Constants and Data Types”, it contains a detailed description of the data types.

With incremental programming, you create the symbol table before entering the program; you can also add or correct individual symbols during program input. In the case of source-oriented programming, the complete symbol table must be available when the program source is compiled.

Importing, exporting

Symbol tables can be imported and exported. “Exported” means a file is created with the contents of your symbol table. You can select here either the entire symbol table, a subset limited by filters or only selected lines. For the data format you can choose between pure ASCII text (extension `*.asc`), sequential assignment list (`*.seq`), System Data Format (`*.sdf` for Microsoft Access) and Data Interchange Format (`*.dif` for Microsoft Excel). You can edit the exported file with a suitable editor. You can also import a symbol table available in one of the formats named above.

Special object properties

With `EDIT → SPECIAL OBJECT PROPERTIES → ...`, you set attributes for each symbol in the symbol table. These attributes or properties are used in the following:

- ▷ Process monitoring with `S7-PDIAG`
- ▷ Human machine interface functions for monitoring with `WinCC`
- ▷ Configuring messages
- ▷ Configuring communications using the `NCM` software
- ▷ Control at contact with inputs and bit memories in the program editor

`VIEW → COLUMNS R, O, M, C, CC` makes the settings visible. With `OPTIONS → CUSTOMIZE`,

you can specify whether or not the special object properties are to be copied and you can define behavior when importing symbols.

2.5.3 Program Editor

For creating the user program, the STEP 7 Basic Package contains a program editor for the LAD, FBD and STL programming languages. You program incrementally with LAD and FBD, that is, you enter an executable block direct; Figure 2.10 shows the possible actions for this.

If you use symbolic addressing for global addresses, the symbols must already be assigned to an absolute address in the case of incremental programming; however, you can enter new symbols or change symbols during program input.

LAD/FBD blocks can be “decompiled”, i.e. a readable block can be created again from the MC7 code without an offline database (you can read any block from a CPU using a programming device without the associated project). In addition, an STL program source can be created from any compiled block.

Starting the program editor

You reach the program editor when you open a block in the SIMATIC Manager, e.g. by double-clicking on the automatically generated symbol of the organization block `OB 1`, or via the Windows taskbar with `START → SIMATIC → STEP 7 → LAD, STL, FBD – PROGRAMMING S7 BLOCKS`.

You can customize the properties of the program editor with `OPTIONS → CUSTOMIZE`. On the “Editor” tab, select the properties with which a new block is to be generated and displayed, such as the creation language, pre-selection for comments, and symbols.

Program editor window

Further windows can be displayed within the window of the program editor: the block window, the *Details* and *Overviews* windows, and the window with the AS registers (Figure 2.11).

The *block window* is automatically displayed when opening a block and contains the block

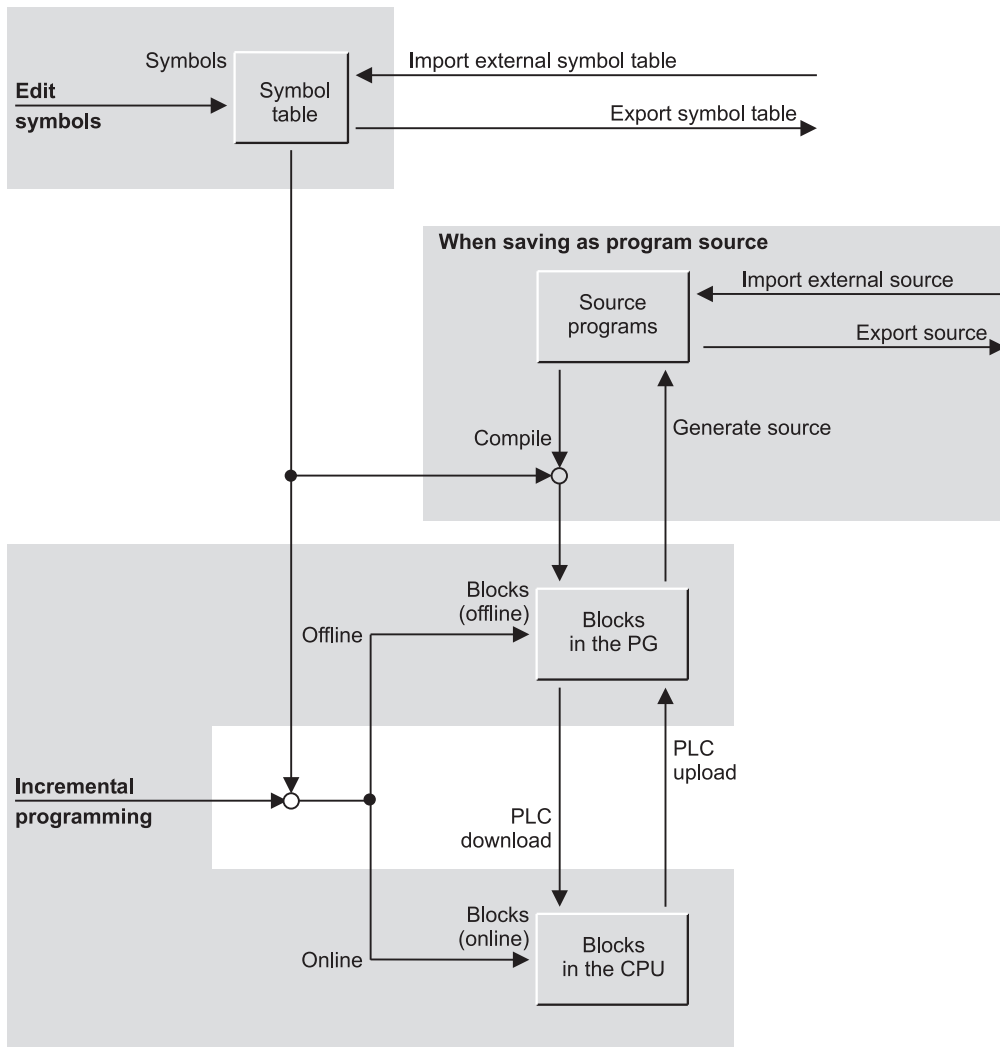


Figure 2.10 Writing Programs with the LAD/FBD Editor

interface at the top, i.e. the block parameters as well as the static and dynamic local data. You can program the block in the bottom program area. The block window and the contents are described in Chapter 3.3.2 “Block Window”.

The *Overviews* window shows the program elements and the call structure. If it is not visible, display it on the screen using **VIEW → OVER-VIEWS**.

The *Details* window can be displayed or suppressed using **VIEW → DETAILS**. It contains the following tabs:

▷ 1: Error

Contains the errors found in the block by the program editor, e.g. following compilation. With **OPTIONS → CUSTOMIZE** in the “Sources” tab you can set whether warnings are also to be displayed.

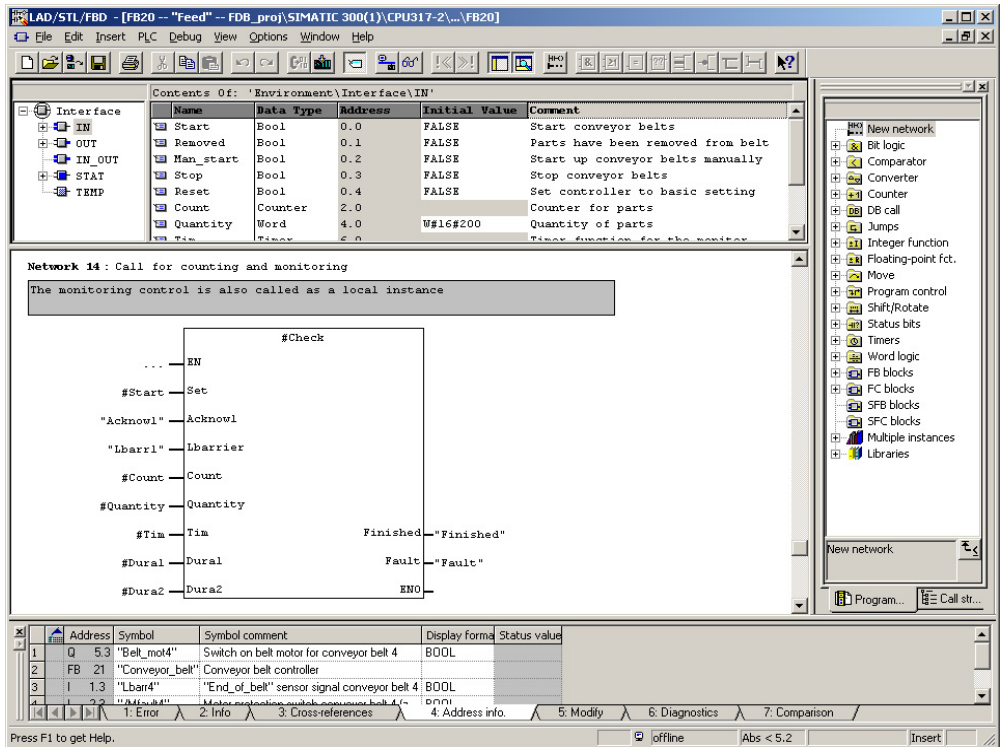


Figure 2.11 Example of editor window

- ▷ 2: Info
Contains information on the currently selected address.
- ▷ 3: Cross-references
Contains the references of addresses present in the current network (see Chapter 2.5.6 “Reference Data”).
- ▷ 4: Address info
Contains the symbol information of the addresses present in the current network (see Chapter 2.5.2 “Symbol Table”). You can edit existing symbols here, and new ones, and observe the address status.
- ▷ 5: Modify
Contains an empty table of variables in which you can enter the addresses to be controlled (see Chapter 2.7.3 “Monitoring and Modifying Variables”).
- ▷ 6: Diagnostics
Contains a list with the existing monitoring

functions for process diagnostics using the S7-PDIAG options package.

- ▷ 7: Comparison
Contains the results of a previously implemented block comparison (see Chapter “Comparing blocks”).

You can dock or undock the *Overviews* and *Details* windows at the edge of the editor window by double clicking the respective title bar.

The *PLC register contents* window shows the contents of the CPU registers (accumulators, address registers and DB registers).

Incremental programming

With incremental programming, you edit the blocks both in the offline and online *Blocks* container. The editor checks your entries in the incremental mode as soon as you have terminated a network. When the block is closed it is

immediately compiled, so that only error-free blocks can be saved.

On the “Block” tab under OPTIONS → CUSTOMIZE, you set automatic updating of the reference data when saving a block.

The blocks can be edited both offline in the programming device's database and online in the CPU, generally referred to as the “programmable controller”, or “PLC”. For this purpose, the SIMATIC Manager provides an offline and an online window; the one is distinguished from the other by the labeling in the title bar.

In the offline window, you edit the blocks right in the PG database. If you are in the editor, you can store a modified block in the offline database with FILE → SAVE and transfer it to the CPU with PLC → DOWNLOAD. If you want to save the opened block under another number or in a different project, or if you want to transfer it to a library or to another CPU, use the menu command FILE → SAVE AS.

With the menu command FILE → STORE READ-ONLY... in the program editor, you can save a write-protected copy of the currently opened (and saved) block in another block container.

To edit a block in the CPU, open that block in the online window. This transfers the block from the CPU to the programming device so that it can be edited. You can write the edited block back to the CPU with PLC → DOWNLOAD. If the CPU is in RUN mode, the CPU will process the edited block in the next program scan cycle. If you want to save a block that you edited online in the offline database as well, you can do so with FILE → SAVE.

With incremental programming, you can execute all programming functions with one exception: if you want to provide block protection (KNOW_HOW_PROTECT), you can only do this via a program source file (see Chapter 24.1 “Block Protection” for more detailed information on this topic).

Chapter 2.6.4 “Loading the User Program into the CPU” and Chapter 2.6.5 “Block Handling” contain further information on online programming. Chapter 3.3 “Programming Code Blocks” and Chapter 3.4 “Programming Data Blocks” show you how to enter a LAD/FBD block.

Decompilation

When the program editor opens a compiled block, it carries out a “decompilation” into the LAD/FBD representation. It uses the program components not relevant to execution in the PG data management in order to display e.g. symbols, comments and jump labels. If the information from the PG data management is missing during the decompilation, the editor uses replacement symbols.

Networks which cannot be decompiled in LAD or FBD are displayed in STL.

Updating or generating source files

On the “Sources” tab under OPTIONS → CUSTOMIZE, you can select the option “Generate source automatically” so that when you save an (incrementally created) block, the program source file is updated or created, if it does not already exist.

You can derive the name of a new source file from the absolute address or the symbolic address. The addresses can be transferred in absolute or symbolic form to the source file.

With the “Execute” button, you select, in the subsequent dialog box, the blocks from which you want to generate a program source file.

With FILE → GENERATE SOURCE you can produce ASCII source files from compiled blocks. First insert a *Sources* container under the object *S7 program*. When generating the source, first enter the storage location and the name of the source in the displayed dialog box, and subsequently select the blocks.

You can export source files from the project by selecting EDIT → EXPORT SOURCE in the SIMATIC Manager. You can then further process these ASCII files with another text editor, for example. Source files can also be imported back into the *Sources* container with INSERT → EXTERNAL SOURCE.

If you generate a source file from a block that you have created with LAD or FBD, you can generate a LAD or FBD block again from this source file. You compile the source file by opening it in the SIMATIC Manager with a double-click and by then selecting FILE → COMPILE in the program editor. An STL block is created in the *Blocks* container. You open this

block and switch to your usual representation with VIEW → LAD or VIEW → FBD. After saving, the block retains this property.

If you selected the setting “Addresses – Symbolic” when creating the source file, you require a complete symbol table for compiling the source file. In this way, you can specify different absolute addresses in the symbol table and, after compilation, you end up with a program with, for example, different inputs and outputs. This allows you to adapt the program to a different hardware configuration. For this purpose, it is best to store these source files (which are independent of the hardware addressing) in a library, for example.

Comparing blocks

The block comparison enables you to find the differences between two blocks. The blocks can be present in different projects, in different target systems (CPUs), or in one project and one target system.

Use the program editor to compare the opened block with the same block in the CPU or in the project by using OPTIONS → COMPARE ON/OFFLINE PARTNER. The result is displayed in the detail area of the editor window in the tab “7: Comparison”.

Mark the *Blocks* object in the SIMATIC Manager, or only the blocks to be compared, and select OPTIONS → COMPARE BLOCKS. The comparison is carried out either between the online and offline data management (ONLINE/offline) or between two projects (Path1/Path2). When comparing the complete program – which can also include tables of variables and user data types (UDTs) – you can incorporate the system data. When using “Execute code comparison”, the program code of the blocks is compared in addition, even of blocks with different generation languages.

The comparison includes all data of a block, even its time stamp for program code and interface. If you wish to know whether the program code is identical independent of the block properties, compare the checksum of the block. To do this, select the “Details” button in the results window of the block comparison.

2.5.4 Rewiring

The *Rewiring* function allows you to replace addresses in individually compiled blocks or in the entire user program. For example, you can replace input bits I 0.0 to I 0.7 with input bits I 16.0 to I 16.7. Permissible addresses are inputs, outputs, memory bits, timers and counters as well as functions FCs and function blocks FBs.

In the SIMATIC Manager, you select the objects in which you wish to carry out the rewiring; select a single block, a group of blocks by holding Ctrl and clicking with the mouse, or the entire *Blocks* user program. OPTIONS → REWIRE takes you to a table in which you can enter the old addresses to be replaced and the new addresses. When you confirm with “OK”, the SIMATIC Manager then exchanges the addresses.

When “rewiring” blocks, change the numbers of the blocks first and then execute rewiring that changes the calls correspondingly. If you “rewire” a function block, its instance data block is automatically assigned to the rewired function block; the data block number is not changed.

Following rewiring, an info file shows you in which block changes were made, and how many.

The reference data are no longer up-to-date following rewiring, and must be regenerated.

Please note that “rewiring” only takes place in the compiled blocks; a program source, if present, is not modified.

Further possible methods of rewiring are:

- ▷ With compiled blocks, you can also use the *Address priority* function.
- ▷ If there is a program source file with symbolic addressing, you change the absolute addresses in the symbol table. Following the compilation, you get an “unwired” program.

2.5.5 Address Priority

In the properties window of the offline object container *Blocks* on the “Address priority” tab, you can set whether the absolute address or the symbol is to have priority for already saved blocks when they are displayed and saved again

following a change to the symbol table or to the declaration or assignment of global data blocks.

The default is “Absolute value has priority” (the same behavior as in the previous STEP 7 versions). This default means that when a change is made in the symbol table, the absolute address is retained in the program and the symbol changes accordingly. If “Symbol has priority” is set, the absolute address changes and the symbol is retained.

Example:

The symbol table contains the following:

```
I 1.0 "Limit_switch_up"
I 1.1 "Limit_switch_down"
```

In the program of an already compiled block, input I 1.0 is scanned:

```
I 1.0 "Limit_switch_up"
```

If the assignments for inputs I 1.0 and I 1.1 are now changed in the symbol table to:

```
I 1.0 "Limit_switch_down"
I 1.1 "Limit_switch_up"
```

and the already compiled block is read out, then the program contains

```
I 1.1 "Limit_switch_up"
```

if “Symbol has priority” is set, and if “Absolute value has priority” is set, the program contains

```
I 1.0 "Limit_switch_down"
```

If, as a result of a change in the symbol table, there is no longer any assignment between an absolute address and a symbol, the statement will contain the absolute address if “Absolute value has priority” is set (even with symbolic display because the symbol would, of course, be missing); if “Symbol has priority” is set, the statement is rejected as errored (because the mandatory absolute address is missing).

If “Symbol has priority” is set, incrementally programmed blocks with symbolic addressing will retain their symbols in the event of a change to the symbol table. In this way, an already programmed block can be “rewired” by changing the address assignment.

Please note that this “rewiring” does not occur automatically because the already compiled blocks contain the executable MC7 code of the statements with absolute addresses. The change is only made in the relevant block – following

the relevant message – after it has been opened and saved again.

In order to carry out the change in the complete block folder, select EDIT → CHECK BLOCK CONSISTENCY with the *Blocks* object marked.

2.5.6 Reference Data

As a supplement to the program itself, the SIMATIC Manager shows you the reference data, which you can use as the basis for corrections or tests. These reference data include the following:

- ▷ Cross references
- ▷ Assignment (Input, Output, Bit Memory, Timers, Counters)
- ▷ Program structure
- ▷ Unused symbols
- ▷ Addresses without symbols

To generate reference data, select the *Blocks* object and the menu command OPTIONS → REFERENCE DATA → DISPLAY. The representation of the reference data can be changed specifically for each work window with VIEW → FILTER; you can save the settings for later editing by selecting WINDOW → SAVE ARRANGEMENT. You can display and view several lists at the same time (Figure 2.12).

With OPTIONS → CUSTOMIZE in the program editor, specify on the “Blocks” tab whether or not the reference data are to be updated when compiling a program source file or when saving an incrementally written block.

Please note that the reference data are only available when the data are managed offline; the offline reference data are displayed even if the function is called in a block opened online.

Cross references

The cross-reference list shows the use of the addresses and blocks in the user program. It includes the absolute address, the symbol (if any), the block in which the address was used, how it was used (read or write) and the positions of use of the address. Click on a column header to sort the table by column contents.

EDIT → Go to → Location with the position marked or a double click on the position starts

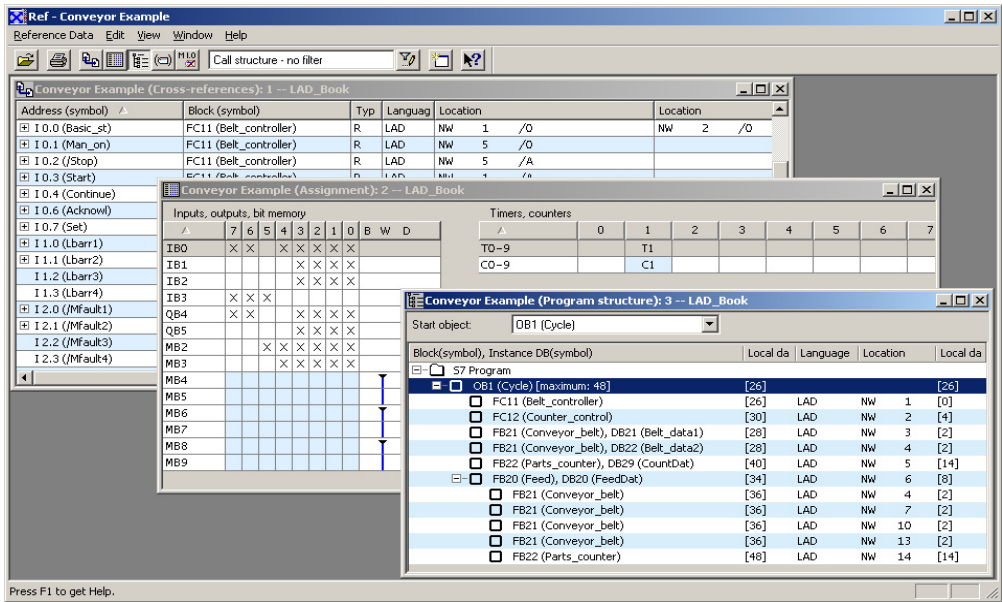


Figure 2.12 Examples of reference data (cross-references, assignment, program structure)

the program editor and displays the address in the programmed environment.

The cross-reference list shows the addresses you selected with VIEW → FILTER (for instance bit memory). STEP 7 then uses the filter saved as "standard" every time it opens the cross-reference list.

Advantage: the cross references show you whether the referenced addresses were also scanned or reset. They also show you in which blocks addresses are used (possibly more than once).

Assignments

The I/Q/M reference list shows which bits in address areas I, Q and M are assigned in the program. One byte, broken down into bits, appears on each line. Also shown is whether access is by byte, word, or doubleword. The T/ C reference list shows the timers and counters used in the program. Ten timers or counters are displayed on a line.

Advantage: the list shows you whether certain address areas were (improperly) assigned or where there are still addresses available.

Program structure

The program structure shows the call hierarchy of the blocks in a user program. You can determine the start object for the call hierarchy from a selection list. With VIEW → FILTER you have a choice between two different views in the program structure:

The *Call structure* shows all nesting levels of the block calls. You control the display of nesting levels with the "+" and "-" boxes. The requirements for temporary local data are shown for one block or for the entire path up to the associated block. With the block selected, change using EDIT → GO TO → LOCATION to the block call, or open the block using EDIT → GO TO → BLOCK ADDRESS.

The display as *Dependency structure* shows two call levels. The blocks are shown (indented) in which the block position on the left is called.

Advantage: Which blocks were used? Were all programmed blocks called? What are the blocks' temporary local data requirements? Is the specified local data requirement per priority class (per organization block) sufficient?

Unused symbols

This list shows all addresses which have symbol table allocations but were not used in the program. The list shows the symbol, the address, the data type, and the comment from the symbol table.

Advantage: were the addresses in the list inadvertently forgotten when the program was being written? Or are they perhaps superfluous, and not really needed?

Addresses without symbol

This list shows all the addresses used in the program to which no symbols were allocated. The list shows these addresses and how often they were used.

Advantage: were addresses used inadvertently (by accident, or because of a typing error)?

2.5.7 Language Setting

STEP 7 offers several methods of working with different languages:

- ▷ The language of the operating system (character set)
- ▷ The STEP 7 language
- ▷ The language for comments and display text

The settings of the different languages are independent of each other.

Language settings in the operating system

You use the Windows control panel to select the character set with which you want to work under Windows. You can find the character sets tested with the multi-language version (MUI version) and the restrictions when operating with STEP 7 in the current Readme file or in the STEP 7 Help under “Setting up and editing the project”.

Project language

The project language is the language that is set in the Windows control panel when the project is created. The SIMATIC Manager indicates the language in which the selected project or the selected library has been created in EDIT → OBJECT PROPERTIES. “Not yet defined” means you can use the project or the library language-neu-

trally, e.g. in multiprojects. These are always language-neutral. In language-neutral projects or libraries, only characters of the ASCII character set can be used (2A_{hex} to 7F_{hex}). You can find additional information in the STEP 7 Help under “Setting up and editing the project”.

STEP 7 language

The session language of the SIMATIC Manager that defines, for example, the menu names and the error messages, is called the STEP 7 language. You set this language in the SIMATIC Manager on the “Language” tab with OPTIONS → CUSTOMIZE. The languages installed with STEP 7 are offered for selection under “National language”. On this tab you also set the programming mnemonics, that is, the language in which STEP 7 uses the addresses and operations, e.g. “A I” (AND input) for English.

Multilingual Comments and Display Texts

Comments and display texts can be multilingual. You have entered the texts in the original language, such as English, and you want to generate a German version of your program. To do so, export the desired texts or text types. The export file is a *.csv file that you can edit with Microsoft Excel, for example. You can enter the translation for each text. You import the finished translation table back into your project. Now you can switch between the languages. You can do this with several languages.

With OPTIONS → Language for display terminals in the SIMATIC Manager, you select the languages available in your project, and you set the standard language for the display terminals.

Exporting and importing texts

Select the object in the SIMATIC Manager containing the comments you want to translate, e.g. the symbol table, the block container, several blocks or a single block. Select OPTIONS → MANAGE MULTILINGUAL TEXTS → EXPORT. In the dialog window that then appears, enter the storage location of the export file and the target language. Select the text types that you want to translate (Table 2.2).

A separate file is generated for every text type, e.g. the file SymbolComment.csv for the comments from the symbol table. Existing export

files can be expanded. A log file provides information on the exported types of text, and any errors which have occurred.

Open the export file(s) with the FILE → OPEN dialog box in Microsoft Excel (not by double-clicking). The exported texts are displayed in the first column and you can translate the texts in the second column.

You can fetch the translated texts back to the project with OPTIONS → MANAGE MULTILINGUAL TEXTS → IMPORT. A log file provides information about the imported texts and any errors that may have occurred.

Please note that the name of the import file must not be changed since there is a direct relation between this and the text types contained in the file.

Selecting and deleting a language

You can change to all imported languages in the SIMATIC Manager with OPTIONS → MANAGE MULTILINGUAL TEXTS → CHANGE LANGUAGE. The language change is executed for the objects (blocks, symbol table) for which the relevant texts have been imported. This information is contained in the log file. You can make further settings, e.g. the “taking over” of multilingual comments when copying a block, using OPTIONS → MANAGE MULTILINGUAL TEXTS → SETTINGS FOR COMMENT MANAGEMENT. You can delete the imported language again with OPTIONS → MANAGE MULTILINGUAL TEXTS → DELETE LANGUAGE.

Table 2.2

Text types of the translated texts (selection)

Text type	Meaning
BlockTitle	Block title
BlockComment	Block comment
NetworkTitle	Network title
NetworkComment	Network comment
LineComment	STL line comment
InterfaceComment	Comment in <ul style="list-style-type: none"> ▷ the declaration table of code blocks ▷ data blocks ▷ user data types UDT
SymbolComment	Symbol comment

2.6 Online Mode

You create the hardware configuration and the user program on the programming device, generally referred to as the “engineering system” (ES). The S7 program is stored offline on the hard disk here, also in compiled form.

To transfer the program to the CPU, you must connect the programming device to the CPU. You establish an “online” connection. You can use this connection to determine the operating state of the CPU and the assigned modules, i.e., you can carry out diagnostics functions.

2.6.1 Connecting a PLC

The connection between the programming device's MPI interface and the CPU's MPI interface is the mechanical requirement for an online connection. The connection is unique when a CPU is the only programmable module connected. If there are several CPUs in the MPI subnet, each CPU must be assigned a unique node number (MPI address). You set the MPI address when you initialize the CPU. Before linking all the CPUs to one network, connect the programming device to only one CPU at a time and transfer the *System Data* object from the offline user program *Blocks* or direct with the Hardware Configuration editor using the menu command PLC → DOWNLOAD. This assigns a CPU its own special MPI address (“naming”) along with the other properties.

The MPI address of a CPU in the MPI network can be changed at any time by transferring a new parameter data record containing the new MPI address to the CPU. Note carefully: the new MPI address takes effect immediately. While the programming device adjusts immediately to the new address, you must adapt other applications, such as global data communications, to the new MPI address.

The MPI parameters are retained in the CPU even after a memory reset. The CPU can thus be addressed even after a memory reset.

A programming device can always be operated online on a CPU, even with a module-independent program and even though no project has been set up.

If no project has been set up, you establish the connection to the CPU with PLC → DISPLAY ACCESSIBLE NODES. This screens a project window with the structure “*Accessible Nodes*” – “Module (MPI=n)” – “Online User Program (Blocks)”. When you select the *Module* object, you may utilize the online functions, such as changing the operational status and checking the module status. Selecting the *Blocks* object displays the blocks in the CPU’s user memory. You can then edit (modify, delete, insert) individual blocks.

You can fetch back the system data from a connected CPU for the purpose of, say, continuing to work on the basis of the existing configuration, without having the relevant project in the programming device data management system. Create a new project in the SIMATIC Manager, select the project and then PLC → UPLOAD STATION TO PG. After specifying the desired CPU in the dialog box that then appears, the online system data are loaded onto the hard disk.

If there is a **CPU-independent program** in the project window, create the associated online project window. If several CPUs are connected to the MPI and accessible, select EDIT → OBJECT PROPERTIES with the online S7 program selected and set the number of the mounting rack and the CPU’s slot on the “Addresses Module” tab.

If you select the *S7 Program* in the online window all the online functions to the connected CPU are available to you. *Blocks* shows the blocks located in the CPU’s user memory. If the blocks in the offline program agree with the blocks in the online program you can edit the blocks in the user memory with the information from the data management system of the programming device (symbolic address, comments).

When you switch a **CPU-assigned program** into online mode using VIEW → ONLINE, you can carry out program modifications just as you would in a CPU-independent program. In addition, it is now possible for you to configure the SIMATIC station, that is, to set CPU parameters and address and parameterize modules.

2.6.2 Protecting the User Program

With appropriately equipped CPUs, access to the user program can be protected with a password. Everyone in possession of the password has unrestricted access to the user program. For those who do not know the password, you can define 3 protection levels. You set the protection levels with the “Protection” tab of the Hardware Configuration tool when parameterizing the CPU.

The access privilege using the password applies until the SIMATIC Manager has been exited or the password protection canceled again using PLC → ACCESS RIGHTS → CANCEL.

Protection level 1: mode selector switch

This protection level is set as default (without password). With CPUs with a keylock switch, protection level 1 is used to set protection of the user program by the mode selector switch on the front of the CPU. In the RUN-P and STOP positions, you have unrestricted access to the user program; in the RUN position, only read access via the programming device is possible. In this position, you can also remove the keylock switch so that the mode can no longer be changed via the switch.

You can bypass protection via the keylock switch RUN position by selecting the option “Removable with password”, e.g. if the CPU, and with it the keylock switch, are not easily accessible or are located at a distance.

If the mode selector switch is designed as a toggle switch, protection level 1 means no limitation in access to the user program.

With the system function SFC 109 PROTECT, the write protection (protection level 2) can be switched on and off via the program in protection level 1 (see Chapter 20.3.8 “Changing program protection”).

Protection level 2: write protection

At this protection level, the user program can only be read, regardless of the position of the keylock switch.

Protection level 3: read/write protection

No access to the user program, regardless of the keylock switch position. Exception: reading of diagnostics buffer and monitoring of variables in tables is possible in every protection level.

Password protection

If you select protection level 2 or 3 or protection level 2 with “Removable with password”, you will be prompted to define a password. The password can be up to 8 characters long.

If you try to access a user program that is protected with a password, you will be prompted to enter the password. Before accessing a protected CPU, you can also enter the password via PLC → ACCESS RIGHTS → SETUP. First, select the relevant CPU or the S7 program.

In the “Enter Password” dialog box, you can select the option “Use password as default for all protected modules” to get access to all modules protected with the same password.

Password access authorization remains in force until the last S7 application has been terminated.

Everyone in possession of the password has unrestricted access to the user program in the CPU regardless of the protection level set and regardless of the keylock position.

2.6.3 CPU Information

In online mode, the CPU information listed below is available to you. The menu commands are screened when you have selected a module (in online mode and without a project) or S7 program (in the online project window).

- ▷ PLC → DIAGNOSTIC/SETTING
 - HARDWARE DIAGNOSTICS
(see Chapter 2.7.1 “Diagnosing the Hardware”)
 - MODULE INFORMATION
General information (such as version), diagnostics buffer, memory (current map of work memory and load memory, compression), cycle time (length of the last, longest, and shortest program cycle), timing system (properties of the CPU clock, clock synchronization, run-time meter), performance

data (available organization blocks and system blocks, sizes of the address areas), communication (data transfer rate and communication links), stacks in STOP state (B stack, I stack, and L stack)

→ OPERATING MODE

Display of the current operating mode (for instance RUN or STOP), modification of the operating mode

→ CLEAR/RESET

Resetting of the CPU in STOP mode

→ SET TIME OF DAY

Setting of the internal CPU clock and - in the enhanced dialog - the difference from a time zone

▷ PLC → CPU MESSAGES

Reporting of asynchronous system errors and of user-defined messages generated in the program with SFC 52 WR_USMSG, SFC 18 ALARM_S, SFC 17 ALARM_SQ, SFC 108 ALARM_D and SFC 107 ALARM_DQ.

▷ PLC → DISPLAY FORCE VALUES, PLC → MONITOR/MODIFY VARIABLES
(see Chapters 2.7.3 “Monitoring and Modifying Variables” and 2.7.4 “Forcing Variables”)

2.6.4 Loading the User Program into the CPU

When you transfer your user program (compiled blocks and configuration data) to the CPU, it is loaded into the CPU’s load memory. Physically, load memory can be a memory integrated in the CPU, a memory card, or a micro memory card (see Chapter 1.1.6 “CPU Memory Areas”).

With a micro memory card or a flash EPROM memory card you can write the card in the programming device and use it as data medium. You plug the card into the CPU in the off-circuit state; on power up following memory reset, the relevant data of the card are transferred to the work memory of the CPU. With appropriately equipped CPUs, you can also overwrite a flash EPROM memory card or a micro memory card if it is plugged into the CPU, but only with the entire program.

In the case of a RAM load memory (integrated in the CPU, as a memory card or micro memory card), you transfer a complete user program by switching the CPU to the STOP state, performing memory reset and transferring the user program. The configuration data are also transferred.

If you only want to change the configuration data (CPU properties, the configured connections, GD communications, module parameters, and so on), you need only load the *System Data* object into the CPU (select the object and transfer it with menu command PLC → DOWNLOAD). The parameters for the CPU go into effect immediately; the CPU transfers the parameters for the remaining modules to those modules during startup.

Please note that the entire configuration is loaded onto the PLC with the *System data* object. If you use PLC → Download... in an application, e.g. in global data communications, only the data edited by the application is transferred.

Note: select PLC → SAVE TO MEMORY CARD to load the compressed archive file (see Chapter 2.2.2 “Managing, Reorganizing and Archiving”). The project in the archive file cannot be edited direct either with the programming device or from the CPU.

2.6.5 Block Handling

Transferring blocks

In the case of a RAM load memory, you can also modify, delete or reload individual blocks in addition to transferring the entire program online.

You transfer individual blocks to the CPU by selecting them in the offline window and selecting PLC → DOWNLOAD. With the offline and online windows opened at the same time, you can also drag the blocks with the mouse from one window and drop them in the other.

Special care is needed when transferring individual blocks during operation. If blocks that are not available in the CPU memory are called within a block, you must first load the “lower-level” blocks. This also applies for data blocks whose addresses are used in the loaded block.

You load the “highest-level” block last. Then, provided it is called, it will be executed immediately in the next program cycle.

Modifying or deleting blocks online

You can edit blocks incrementally with STL in the online user program (on the CPU) in exactly the same way as in the offline user program. With a programming device connected online to the CPU, you can read, modify or delete blocks in the load memory.

If the RAM component of the load memory is large enough to accommodate the complete user program and also the modified blocks, you can edit blocks without limitation.

If the user program is saved on a flash EPROM memory card, you can edit the blocks as long as the RAM component of the load memory is large enough to accommodate the modified blocks. During runtime, the modified blocks in the RAM are valid, those in the FEPRM as invalid. Please note that you must load the original blocks again from the FEPRM into the work memory following an overall reset or non-buffered switching on.

If you use a micro memory card such as e.g. with the compact CPUs, all blocks in the load memory are non-volatile. You can modify individual blocks online, and these blocks retain their changes even following an overall reset or non-buffered switching on. Deleted blocks are then no longer existent.

In incremental programming mode, you can modify blocks independent of one another in the offline data management on the programming device and in the online data management on the CPU. However, if online and offline data management diverge, it may result in the editor being unable to display the additional information of the offline database; these data can then be lost (symbolic identifiers, jump labels, comments, user data types).

Blocks that have been modified online are best stored offline on the hard disk to avoid data inconsistency (e.g. a “time stamp conflict” when the interface of the called block is later than the program in the calling block).

The following still applies even if you work online: with FILE → SAVE you save the current block in the offline user program in the PG data management; with PLC → DOWNLOAD you write the block back into the user memory and the CPU.

Compressing

When you load a new or modified block into the CPU, the CPU places the block in load memory and transfers the relevant data to work memory. If there is already a block with the same number, this “old block” is declared invalid (following a prompt for confirmation) and the new block “added on at the end” in memory. Even a deleted block is “only” declared invalid, not actually removed from memory.

This results in gaps in user memory which increasingly reduce the amount of memory still available. These gaps can be filled only by the *Compress* function. When you compress in RUN mode, the blocks currently being executed are not relocated; only in STOP mode can you truly achieve compression without gaps.

The current memory allocation can be displayed in percent with the menu command PLC → DIAGNOSTIC/SETTING → MODULE INFORMATION, on the “Memory” tab. The dialog box which then appears also has a button for preventive compression.

You can initiate event-driven compressing per program with the call SFC 25 COMPRESS.

Data blocks offline/online

During programming, you assign the data addresses in a data block with a default value and an initial value (see also Chapter 3.4 “Programming Data Blocks”). If a data block is loaded into the CPU, the initial values are transferred to load memory and subsequently to work memory, where they become actual values. Every value change made to a data address per program corresponds to a change to the actual value in the work memory (Figure 2.13).

You can upload the actual values generated in the work memory from a programmed (loaded) data block into the offline data management by opening the data block online and transferring it

in the program editor to the offline data management using FILE → SAVE. The names of the variables and the data types saved in the offline data management are then retained. If you upload the online data block in the SIMATIC Manager using PLC → UPLOAD TO PG or by dragging the data block from the online window to the offline window in the offline data management, the description of the addresses, such as e.g. the name of the variables or the data type, is lost.

If you upload a data block from the CPU back into the offline data management, the actual values present in the work memory are imported into the offline data management as initial values. This does not change the initial values in the load memory. Following an overall reset or non-buffered switching on, and when using a flash EPROM memory card or a micro memory card, the (old) initial values present in the load memory are imported into the work memory as (new) actual values.

If you wish to import the actual values into the load memory when using a RAM load memory or a micro memory card, load the data block from the CPU into the programming device and then back again into the CPU. CPUs with micro memory card provide the system function SFC 84 WRIT_DBL with which you can directly write actual values into the load memory. With correspondingly designed CPUs, you can transfer the complete contents of the work memory into the ROM component of the load memory using PLC → COPY RAM TO ROM.

A data block generated with the property *Unlinked* is not transferred to work memory; it remains in load memory. A data block with this property can only be read with SFC 20 BLK-MOV or – with correspondingly designed CPUs – with SFC 83 READ_DBL.

In incremental programming mode, you can create data blocks directly in the work memory of the CPU. It is recommendable to also save these blocks offline immediately they have been created.

With the system functions SFC 22 CREAT_DB, SFC 85 CREA_DB and SFC 82 CREA_DBL you can generate data blocks during runtime where the description of the addresses, such as the name of the variable and the type of

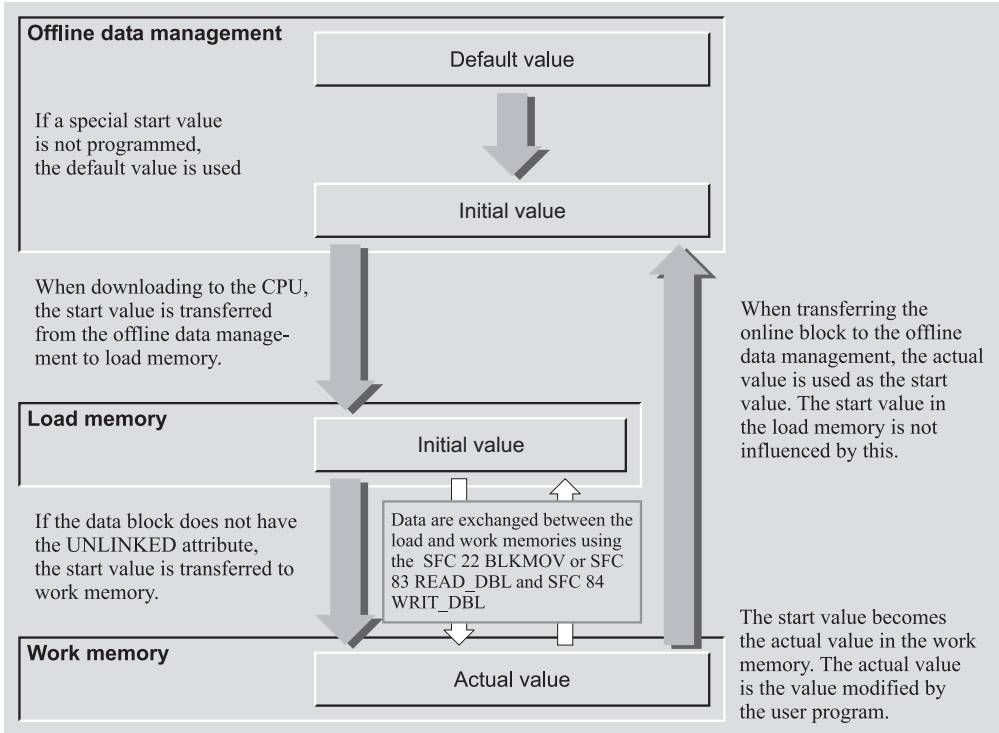


Figure 2.13 Data storage in user memory

data, are missing. When reading with the programming device, a BYTE field with a name and index assigned by the program editor is therefore displayed. If you transfer such a data block to the offline data management, this declaration is imported. If the data block has the property *Unlinked*, the initial values from the load memory are imported into the offline data management as new initial values, otherwise the actual values from the work memory. When transferring to the offline data management, the checksum of the (offline) program is changed.

2.7 Testing the Program

After establishing a connection to a CPU and loading the user program, you can test (debug) the program as a whole or in part, such as individual blocks. You initialize the variables with signals and values, e.g. with the help of simulator modules and evaluate the information

returned by your program. If the CPU goes to the STOP state as a result of an error, you can get support in finding the cause of the error from the CPU information among other things.

Extensive programs are debugged in sections. If, for example, you only want to debug one block, load this block into the CPU and call it in OB 1. If OB 1 is organized in such a way that the program can be debugged section by section "from beginning to end", you can select the blocks or program sections for debugging by using jump functions to skip those calls or program sections that are not to be debugged.

With the S7 PLCSIM optional software, you can simulate a CPU on the programming device and so debug your program without additional hardware.

2.7.1 Diagnosing the Hardware

In the event of a fault, you can fetch the diagnostics information of the faulty modules with

the help of the function “Diagnose Hardware”. You connect the programming device to the MPI bus and start the SIMATIC Manager.

If the project associated with the plant configuration is available in the programming device database, open the online project window with VIEW → ONLINE. Otherwise, select PLC → DISPLAY ACCESSIBLE NODES in the SIMATIC Manager, and select the CPU.

Now you can get a quick overview of the faulty modules with PLC → DIAGNOSTIC/SETTING → HARDWARE DIAGNOSTICS (default in the SIMATIC Manager under OPTIONS → CUSTOMIZE in the tab “View”). If the quick overview is deselected, you are provided with the detailed diagnostics information of all modules.

If you are in the Hardware Configuration tool, select the online view using VIEW → ONLINE. You can now display the existing diagnostics information for the selected module using PLC → MODULE INFORMATION.

2.7.2 Determining the Cause of a STOP

If the CPU goes to STOP because of an error, the first measure to take in order to determine the reason for the STOP is to output the diagnostics buffer. The CPU enters all messages in the diagnostic buffer, including the reason for a STOP and the errors which led to it.

To output the diagnostic buffer, switch the PG to online, select an S7 program, and choose the “Diagnostics Buffer” tab with the menu command PLC → DIAGNOSTIC/SETTING → MODULE INFORMATION. The last event (the one with the number 1) is the cause of the STOP, for instance “STOP because programming error OB not loaded”. The error which led to the STOP is described in the preceding message, for example “FC not loaded”. By clicking on the message number, you can screen an additional comment in the next lower display field. If the message relates to a programming error in a block, you can open and edit that block with the “Open Block” button.

If the cause of the STOP is, for example, a programming error, you can ascertain the surrounding circumstances with the “Stacks” tab. When you open “Stacks”, you will see the B stack (block stack), which shows you the call

path of all non-terminated blocks up to the block containing the interrupt point. Use the “I stack” button to screen the interrupt stack, which shows you the contents of the CPU registers (accumulators, address register, data block register, status word) at the interrupt point at the instant the error occurred. The L stack (local data stack) shows the block's temporary local data, which you select in the B stack by clicking with the mouse.

2.7.3 Monitoring and Modifying Variables

One excellent resource for debugging user programs is the monitoring and modifying of variables with VAT variable tables. Signal states or values of variables of elementary data types can be displayed. If you have access to the user program, you can also modify variables, i.e. change the signal state or assign new values.

Please note that you can only modify data addresses if the write protection for the data block is switched off, i.e. the block property *DB is write-protected in the AS* is not activated.

Operands in data blocks with the block property *Unlinked* cannot be monitored since these data blocks are located in the load memory on the micro memory card. A one-off update takes place when the data block is opened online.

Caution: you must ensure that no dangerous states can result from modifying variables!

Creating a variable table

For monitoring and modifying variables, you must create a VAT variable table containing the variables and the associated data formats. You can generate up to 255 variable tables (VAT 1 to VAT 255) and assign them names. The maximum size of a variable table is 1024 lines with up to 255 characters (Figure 2.14).

You can generate a VAT offline by selecting the user program *Blocks* and then INSERT → S7 BLOCK → VARIABLE TABLE, and you can generate an unnamed VAT online by selecting *S7 Program* and selecting PLC → MONITOR/MODIFY VARIABLES.

You can specify the variables with either absolute or symbolic addresses and choose the data type (display format) with which a variable is to

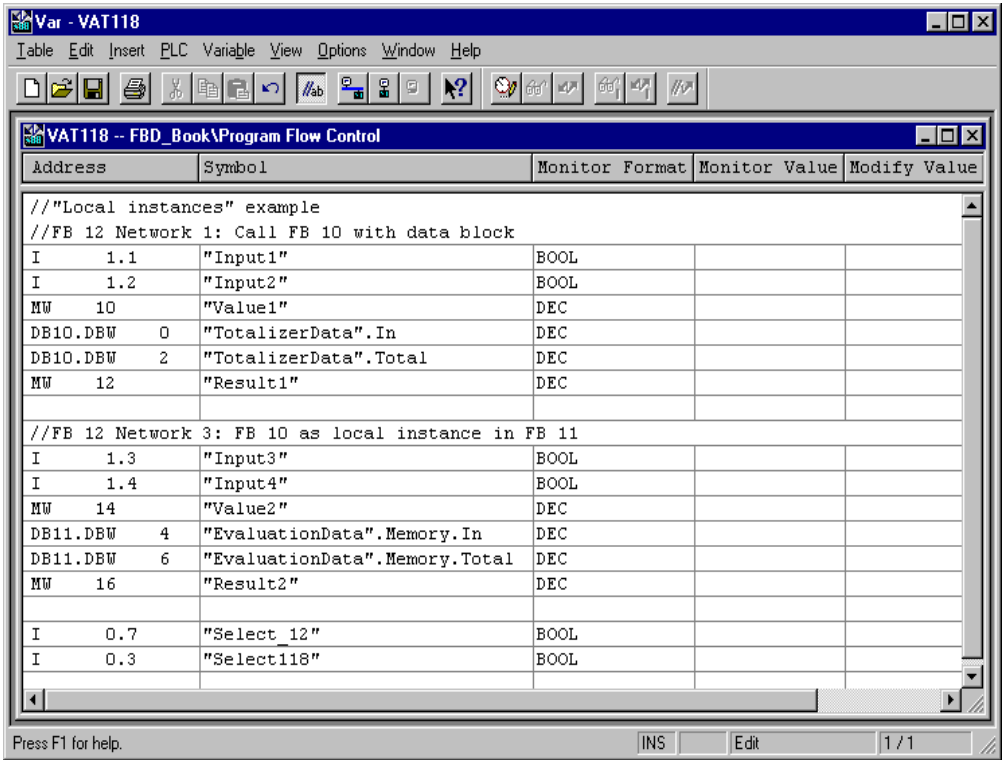


Figure 2.14 Variable Table Example

be displayed and modified (with VIEW → SELECT DISPLAY FORMAT or by clicking the right mouse button directly on the “Display format”).

Use comment lines to give specific sections of the table a header. You may also stipulate which columns are to be displayed. You can change variable or display format or add or delete lines at any time. You save the variable table in the *Blocks* object container with TABLE → SAVE.

Establishing an online connection

To operate a variable table that has been created offline, switch it online with PLC → Connect to → ... online. You must switch each individual VAT online and you can clear down the connection again with PLC → DISCONNECT.

Trigger conditions

In the variable table, select VARIABLE → TRIGGER to set the trigger point and the trigger conditions separately for monitoring and modifying. The trigger point is the point at which the CPU reads values from the system memory or writes values to the system memory. You specify whether reading and writing is to take place once or periodically.

If monitoring and modifying have the same trigger conditions, monitoring is carried out before modifying. If you select the trigger point “Start of cycle” for modifying, the variables are modified after updating of the process input image and before calling OB 1. If you select the trigger point “End of cycle” for monitoring, the status values are displayed after termination of OB 1 and before output of the process output image.

Monitoring variables

Select the Monitor function with the menu command `VARIABLE → MONITOR`. The variables in the VAT are updated in accordance with the specified trigger conditions. Permanent monitoring allows you to follow changes in the values on the screen. The values are displayed in the data format which you set in the “Display format” column. The ESC key terminates a permanent monitor function.

`VARIABLE → Activate modify values` transfers the modify values only once and immediately, without regard to the specified trigger conditions.

Modifying variables

Use `VARIABLE → MODIFY` to transfer the specified values to the CPU dependent on the trigger conditions. Enter values only in the lines containing the variables you want to modify. You can expand the commentary for a value with “//” or with `VARIABLE → MODIFY VALUE AS COMMENT`; these values are not taken into account for modification. You must define the values in the data format which you set in the “Display format” column. Only the values visible on starting the modify function are modified. The ESC key terminates a permanent modify function.

`VARIABLE → ACTIVATE MODIFY VALUES` transfers the modify values only once and immediately, without regard to the specified trigger conditions.

2.7.4 Forcing Variables

With appropriately equipped CPUs, you can specify fixed values for certain variables. The user program can no longer change these values (“forcing”). Forcing is permissible in any CPU operating state and is executed immediately.

Caution: you must ensure that no dangerous states can result from forcing variables!

The starting point for forcing is a variable table (VAT). Create a VAT, enter the addresses to be forced and establish a connection to the CPU. You can open a window containing the force values by selecting `VARIABLE → DISPLAY FORCE VALUES`.

If there are already force values active in the CPU, these are indicated in the force window in bold type. You can now transfer some or all addresses from the variable table to the force window or enter new addresses. You save the contents of a force window in a VAT with `TABLE → SAVE AS`.

The following address areas can be provided with a force value:

- ▷ Inputs I (process image)
[S7-300 and S7-400]
- ▷ Outputs Q (process image)
[S7-300 and S7-400]
- ▷ Peripheral inputs PI
[S7-400]
- ▷ Peripheral outputs PQ
[S7-400]
- ▷ Memory bits M
[S7-400]

You start the force job with `VARIABLE → FORCE`. The CPU accepts the force values and permits no more changes to the forced addresses.

While the force function is active, the following applies:

- ▷ All read accesses to a forced address via the user program (e.g. load) and via the system program (e.g. updating of the process image) always yield the force value.
- ▷ On the S7-400, all write accesses to a forced address via the user program (e.g. transfer) and via the system program (e.g. via SFCs) remain without effect. On the S7-300, the user program can overwrite the force values.

Forcing on the S7-300 corresponds to cyclic modifying: after the process input image has been updated, the CPU overwrites the inputs with the force value; before the process output image is output, the CPU overwrites the outputs with the force value.

Note: forcing is not terminated by closing the force window or the variable table, or by breaking the connection to the CPU! You can only delete a force job with `VARIABLE → STOP FORCING`.

Forcing is also deleted by memory reset or by a power failure if the CPU is not battery-backed. When forcing is terminated, the addresses retain the force values until overwritten by either the user program or the system program.

Forcing is effective only on I/O assigned to a CPU. If, following restart, forced peripheral inputs and outputs are no longer assigned (e.g. as a result of reparameterizing), the relevant peripheral inputs and outputs are no longer forced.

Error handling

If the access width when reading is greater than the force width (e.g. forced byte in a word), the unforced component of the address value is read as usual. If a synchronization error occurs here (access or area length error) the “error substitute value” specified by the user program or by the CPU is read or the CPU goes to STOP.

If, when writing, the access width is greater than the force width (e.g. forced byte in a word), the unforced component of the address value is written to as usual. An errored write access leaves the forced component of the address unchanged, i.e. the write protection is not revoked by the synchronization error.

Loading forced peripheral inputs yields the force value. If the access width agrees with the force width, input modules that have failed or have not (yet) been plugged in can be “replaced” by a force value.

The input I in the process image belonging to a forced peripheral input PI is not forced; it is not preassigned and can still be overwritten. When updating the process image, the input receives the force value of the peripheral input.

When forcing peripheral outputs PQ, the associated output Q in the process image is not updated and not forced (forcing is only effective “externally” to the module outputs). The outputs Q are retained and can be overwritten; reading the outputs yields the written value (not the force value). If an output module is forced and if this module fails or is removed, it will receive the force value again immediately on reconnection.

The output modules output signal state “0” or the substitute value with the OD signal (disable

output modules at STOP, HOLD or RESTART) – even if the peripheral outputs are forced (exception: analog modules without OD evaluation continue to output the force value). If the OD signal is deactivated, the force value becomes effective again.

If, in STOP mode, the function *Enable PQ* is activated, the force values also become effective in STOP mode (due to deactivation of the OD signal). When *Enable PQ* is terminated, the modules are set back to the “safe” state (signal state “0” or substitute value); the force value becomes effective again at the transition to RUN.

2.7.5 Enabling Peripheral Outputs

In STOP mode, the output modules are normally disabled by the OD signal; with the Enable peripheral outputs function, you can deactivate the OD signal so that you can modify the output modules even at CPU STOP. Modifying is carried out via a variable table. Only the peripheral outputs assigned to a CPU are modified. Possible application: wiring test of the output at STOP and without user program.

Caution: you must ensure that no dangerous states can result from enabling the peripheral outputs!

Create a variable table and enter the peripheral outputs (PQ) and the modify values. Switch the variable table online with PLC → CONNECT TO → ... and stop the CPU if necessary, e.g. with PLC → OPERATING MODE and "STOP".

You deactivate the OD signal VARIABLE → ENABLE PERIPHERAL OUTPUTS; the module outputs now have signal state “0” or the substitute value or force value. You modify the peripheral outputs with VARIABLE → ACTIVATE MODIFY VALUES. You can change the modify value and modify again.

You can switch the function off again by selecting VARIABLE → ENABLE PERIPHERAL OUTPUTS again, or by pressing the ESC key. The OD signal is then active again, the module outputs are set to “0” and the substitute value or the force value is reset.

If STOP is exited while “enable peripheral outputs” is still active, all peripheral inputs are deleted, the OD signal is activated at the transi-

tion to RESTART and deactivated again at the transition to RUN.

2.7.6 Test and process operation

The recording of the program status information requires additional execution time in the program cycle. For this reason, you can choose two operating modes for debugging purposes: test mode and process mode. In *test mode*, all debugging functions can be used without restriction. You would select this, for example, to debug blocks without connection to the system, because this can significantly increase the cyclic execution time. In *process mode*, care is taken to keep the increase in cycle time to a minimum and this results in debugging restrictions, e.g. the status display with program loops is aborted at the return point. Debugging and step-by-step program execution cannot be performed in process operation.

Test mode is set in the factory on the S7-300 CPUs. You can set test or process mode on these CPUs with the Hardware Configuration on the “Protection” tab. Following this, the configuration must be compiled again and downloaded to the CPU.

The process mode is set in the factory on the S7-400 CPUs. You can change the operating mode online with the Program Editor. `DEBUG → OPERATION...` displays the set operating mode and offers the facility of changing this online.

2.7.7 LAD/FBD Program Status

With the *Program status* function, the program editor provides an additional test method for the user program. The editor shows you the binary signal flow and digital values within a network.

The block whose program you want to debug is in the CPU's user memory and is called and edited there. Open this block, for example by double-clicking on it in the SIMATIC Manager's online window. The editor is started and shows the program of the block.

Select the network you want to debug. Activate the Program Status function with `DEBUG → MONITOR`. Now you can see the binary signal

flow in the block window and you can follow the changes in it (Figure 2.16). You define the representation (e.g. color) in the program editor with `OPTIONS → CUSTOMIZE` on the “LAD/FBD” tab. You can deactivate the Program Status function again by selecting `DEBUG → MONITOR` again.

You set the trigger conditions with `DEBUG → CALL ENVIRONMENT` with debug mode switched on (see chapter 2.7.6 “Test and process operation”). You require this setting if the block to be debugged is called more than once in your program. You can initiate status recording either by specifying the call path (determined from the reference data or manually) or by making it dependent on the opened data blocks when calling the block to be debugged. If you do not set the call environment, you monitor the block when it is called for the first time.

Modifying addresses

You can modify addresses in the program status function. If the address is of data type `BOOL`, mark it and select `DEBUG → MODIFY ADDRESS TO 0` or `DEBUG → MODIFY ADDRESS TO 1`. With a different data type, select `DEBUG → MODIFY ADDRESS`, and enter the modified value for the marked address in the dialog box displayed.

Operations on the contact

In the Program Status function, you can directly modify binary inputs and bit memories in the user program by means of a button. The following prerequisites exist for this function:

- ▷ In the symbol table, you assign the inputs and bit memories with the attribute `CC` (Control at Contact, see “Special object properties” in Chapter 2.5.2 “Symbol Table”).
- ▷ You have enabled operations on the contact in the program editor using `OPTIONS → CUSTOMIZE` on the “General” tab.
- ▷ You are online in the program status with `DEBUG → MONITOR` and additionally select `DEBUG → CONTROL AT CONTACT`.

The symbols and addresses of the binary inputs and bit memories are displayed as buttons which you can access using the mouse.

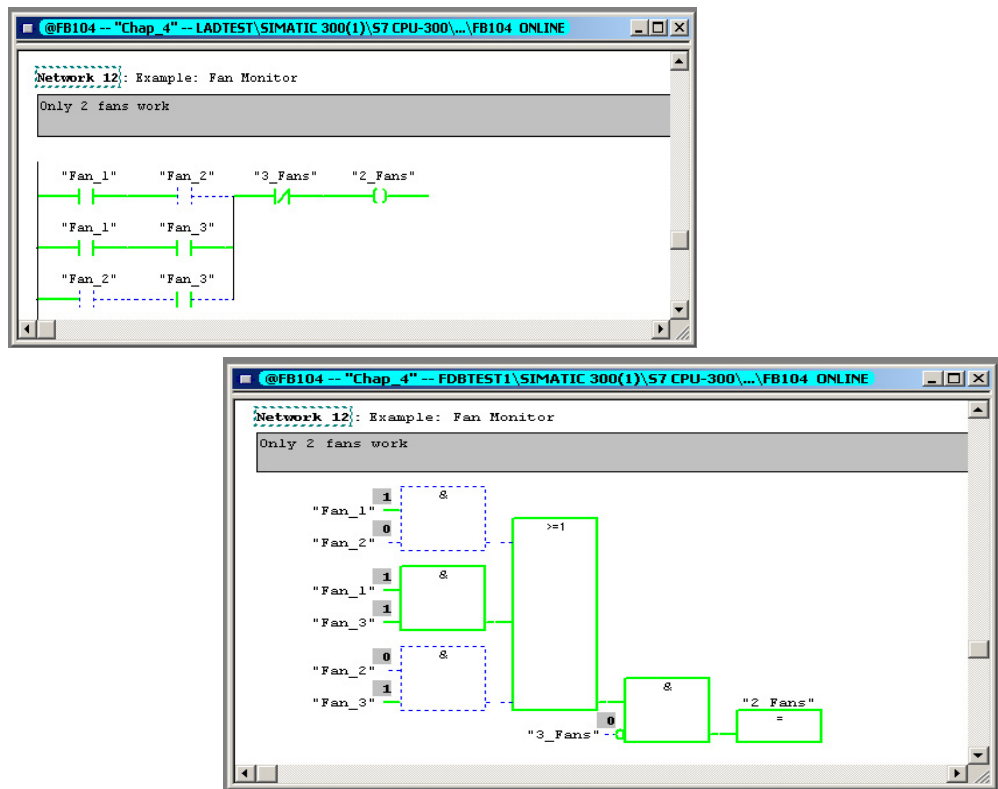


Figure 2.15 LAD/FBD Program Status

Addresses programmed as NO contacts or addresses with scanning for signal status “1” then deliver the address status “1”; addresses programmed as NC contacts or addresses with scanning for signal status “0” deliver the status “0” when accessed. You can use the Ctrl/Strg key and the mouse to select several addresses and to access them simultaneously when operating on the contact. You deselect the operands in the same manner.

2.7.8 Monitoring and Modifying Data Addresses

If the variables to be debugged are present in data blocks, you can also view and modify them directly. Select the data block in the SIMATIC Manager and select EDIT → OPEN OBJECT. With STEP 7 V5.2 and later, you will be asked in the default setting whether you wish to open the data block using the program editor or using

the application “Parameter assignment for data blocks”.

Switch the data view on in the program editor using VIEW → DATA VIEW, and select DEBUG → MONITOR. You can now view the actual values in the work memory, and also set (modify) them if required. Using PLC → DOWNLOAD, download the modified actual values into the work memory, or use FILE → SAVE to import the modified values into the offline data management (first switch off DEBUG → MONITOR).

Using “Parameter assignment for data blocks” you can directly view and modify the actual values in the work memory of the CPU. You can also view the actual values here using DEBUG → MONITOR, and you can also adjust them. Using PLC → DOWNLOAD PARAMETER SETTING DATA you have the possibility for only writing the actual values into the work memory, and not the complete data block. Using DATA

BLOCK → SAVE you can import the data block into the offline data management.

The advantage of the application “Parameter assignment for data blocks” is to be found in the possibility for displaying and parameterizing data blocks in the parameterization view. Pre-requisite: the system attribute *S7-techparam* (technological functions) is set, and a parameterization desktop is available, e.g. from an

option package. Figure 2.16 shows a comparison between parameterization view and data view using an example of the instance data block for the controller function block FB 58 TCONT_CP from the standard library *PID Control Blocks*. Its parameterization desktop is supplied together with STEP 7.

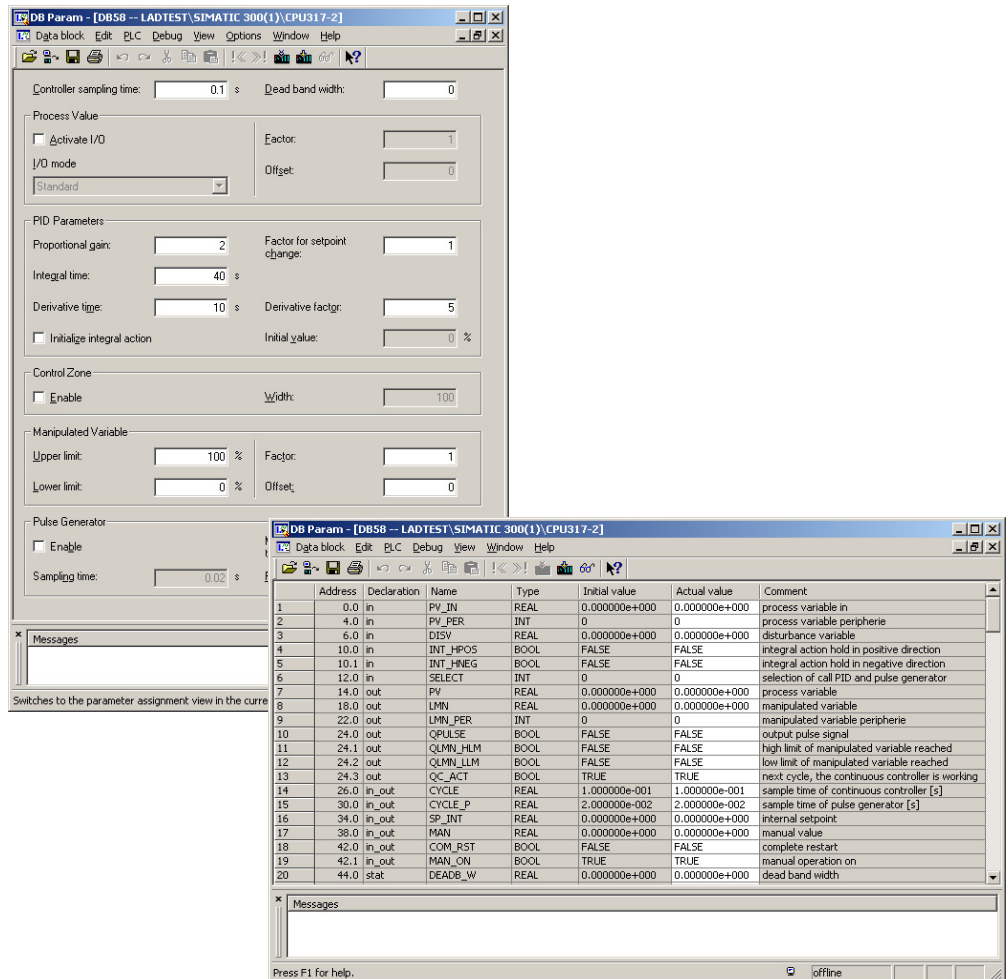


Figure 2.16 Comparison between parameterization view and data view

3 SIMATIC S7 Program

This chapter shows you the structure of the user program for the SIMATIC S7-300/400 CPUs starting from the different priority classes (program execution types) via the component parts of a user program (blocks) right up to the variables and data types. The focus of this chapter is the description of block programming with LAD and FBD. Following this is a description of the data types.

You define the structure of the user program right back at the design phase when you adapt the technological and functional conditions; it is decisive for program creation, program test and startup. To achieve effective programming, it is therefore necessary to devote special attention to the program structure.

3.1 Program Processing

The overall program for a CPU consists of the operating system and the user program.

The operating system is the totality of all instructions and declarations which control the system resources and the processes using these resources, and includes such things as data backup in the event of a power failure, the activation of priority classes, and so on. The operating system is a component of the CPU to which you, as user, have no write access. However, you can reload the operating system from a memory card, for instance in the event of a program update.

The user program is the totality of all instructions and declarations for signal processing, through which a plant (process) is affected in accordance with the defined control task.

3.1.1 Program Processing Methods

The user program may be composed of program sections which the CPU processes in dependence on certain events. Such an event might be the start of the automation system, an interrupt, or detection of a program error (Figure 3.1). The programs allocated to the events are divided into priority classes, which determine the program processing order (mutual interruptibility) when several events occur.

The lowest-priority program is the main program, which is processed cyclically by the CPU. All other events can interrupt the main program at any location, the CPU then executes the associated interrupt service routine or error handling routine and returns to the main program.

A specific organization block (OB) is allocated to each event. The organization blocks represent the priority classes in the user program. When an event occurs, the CPU invokes the assigned organization block. An organization block is a part of a user program which you yourself may write.

Before the CPU begins processing the main program, it executes a startup routine. This routine can be triggered by switching on the mains power, by actuating the mode switch on the CPU's front panel, or via the programming device. Program processing following execution of the startup routine always starts at the beginning of the main program in the case of a cold restart or warm restart; in S7-400 systems, it is also possible to resume the program scan at the point at which it was interrupted (hot restart).

The main program is in organization block OB 1, which the CPU always processes. The start of the user program is identical to the first network in OB 1. After OB 1 has been processed (end of program), the CPU returns to the oper-

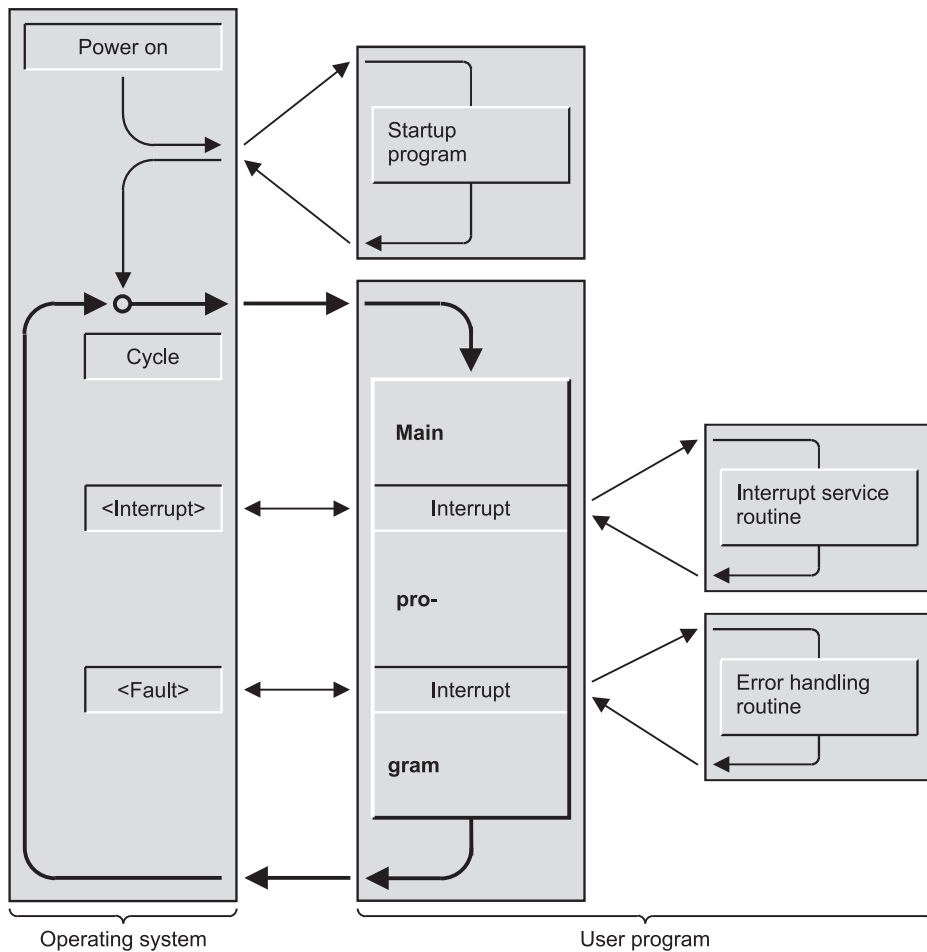


Figure 3.1 Methods of Processing the User Program

ating system and, after calling for the execution of various operating system functions, such as the updating of the process images, it once again calls OB 1.

Events which can intervene in the program are interrupts and errors. Interrupts can come from the process (hardware interrupts) or from the CPU (watchdog interrupts, time-of-day interrupts, etc.). As far as errors are concerned, a distinction is made between synchronous and asynchronous errors.

An asynchronous error is an error which is independent of the program scan, for example failure of the power to an expansion unit or an interrupt that was generated because a module was being replaced.

A synchronous error is an error caused by program processing, such as accessing a non-existent address or a data type conversion error. The type and number of recorded events and the associated organization blocks are CPU-specific; not every CPU can handle all possible STEP 7 events.

3.1.2 Priority Classes

Table 3.1 lists the available SIMATIC S7 organization blocks, each with its priority. In some priority classes, you can change the assigned priority when you parameterize the CPU. The Table shows the lowest and highest possible priority classes; each CPU has a different low/high range; a specific CPU occupies a section of this overview.

Organization block OB 90 (background processing) executes alternately with organization block OB 1, and can, like OB 1, be interrupted by all other interrupts and errors.

The startup routine may be in organization block OB 100 (warm restart), OB 101 (hot restart) or OB 102 (cold restart), and has priority 27. Asynchronous errors occurring in the startup routine have priority class 28. Diagnostic interrupts are regarded as asynchronous errors.

You determine which of the available priority classes you want to use when you parameterize the CPU. Unused priority classes (organization blocks) must be assigned priority 0.

The relevant organization blocks must be programmed for all priority classes used; otherwise the CPU will invoke OB 85 “Program Processing Error” or go to STOP.

For each priority class selected, temporary local data (L stack) must be available in sufficient volumes (see Chapter 18.1.5 “Temporary Local Data” for more details).

3.1.3 Specifications for Program Processing

The CPU’s operating system normally uses default parameters. You can change these defaults when you parameterize the CPU (in the Hardware object) to customize the system to suit your particular requirements. You can change the parameters at any time.

Every CPU has its own specific number of parameter settings. The following list provides an overview of all STEP 7 parameters and their most important settings.

- ▷ General
Name of CPU, plant identifier, location ID, settings for MPI interface (if the interface is not combined with DP), comment
- ▷ Startup
Specifies the type of startup (cold restart, warm restart, hot restart); monitoring of Ready signals or module parameterization; maximum amount of time which may elapse before a warm restart
- ▷ Cycle/Clock Memory
Enable/disable cyclic updating of the process image; specification of the cycle monitoring time and minimum cycle time; amount of cycle time, in percent, for communication; number of the clock memory byte; size of the process images
- ▷ Retentive Memory
Number of retentive memory bytes, timers and counters; specification of retentive areas for data blocks
- ▷ Memory
max. number of temporary local data in the priority classes (organization blocks); max size of the L stack and number of communications jobs
- ▷ Interrupts
Specification of the priority for hardware interrupts, time-delay interrupts, asynchronous errors and DPV1 interrupts; assignment of partial process images with process and time delay interrupts
- ▷ Time-of-Day Interrupts
Specification of the priority and assignments of partial process images; specification of the start time and periodicity
- ▷ Cyclic Interrupts
Specification of the priority, the time cycle and the phase offset; assignment of partial process images
- ▷ Synchronous cycle Interrupts
Specification of the priority; assignment of DP master system and partial process images
- ▷ Diagnostics/Clock
Specification of the system diagnostics; type and interval for clock synchronization, correction factor

- ▷ Protection
Specification of the protection level; defining a password; setting of process or debug mode
 - ▷ Multicomputing
Specification of the CPU number
 - ▷ Integrated functions
Activation and parameterization of the integrated functions
 - ▷ Communications
Definition of connection resources
 - ▷ Web
Activation of the Web server, language selection
- On startup, the CPU puts the user parameters into effect in place of the defaults, and they remain in force until changed.

Table 3.1 SIMATIC S7 Organization Blocks

Organization block	Called	Priority	
		Default	Modifiable
Free cycle OB 1	Cyclically via the operating system	1	No
TOD interrupts OB 10 to OB 17	At a specific time of day or at regular intervals (e.g. monthly)	2	0, 2 to 24
Time-delay interrupts OB 20 to OB 23	After a programmable time, controlled by the user program	3 to 6	0, 2 to 24
Watchdog interrupts OB 30 to OB 38	Regularly at programmable intervals (e.g. every 100 ms)	7 to 15	0, 2 to 24
Process interrupts OB 40 to OB 47	On interrupt signals from I/O modules	16 to 23	0, 2 to 24
DPV1 interrupts OB 55 to OB 57	With status, update or vendor alarms from PROFIBUS DPV1 slaves	2	0, 2 to 24
Multiprocessor interrupt OB 60	Event-driven via the user program in multi-computing	25	No
Synchronous cycle interrupts OB 61 to OB 64	With Synchronous cycle interrupt of DP master (synchronous with DP cycle)	25	0, 2 to 26
Technology sync interrupt OB 65	Synchronous following updating of the technology data blocks of a CPU 317T	25	No
Redundancy error interrupts OB 70,	In the case of loss of redundancy resulting from I/O errors,	25	2 to 26
OB 72,	In the case of CPU redundancy error	28	2 to 28
OB 73	In the case of communications redundancy error	25	2 to 26
Asynchronous error interrupts OB 80 OB 81 to OB 84, 86, 87 OB 85 OB 88	In the case of errors not involved in program execution (e.g. time error, SE error, diagnostics interrupt, insert/remove module interrupt, rack/station failure, processing abort)	26 ²⁾ 25 ²⁾ 25 ²⁾ 28	No 2 to 26 24 to 26 No
Background processing OB 90	Minimum cycle time duration not yet reached	29 ¹⁾	No
Startup routine OB 100, OB 101, OB 102	At programmable controller startup	27	No
Synchronous errors OB 121, OB 122	In the case of errors connected with program execution (e.g. I/O access error)	Priority of the OBs causing the errors	

¹⁾ see text²⁾ at startup: 28

Program length, memory requirements

The memory requirements of a compiled block are listed in the block properties. If you select the block in the SIMATIC Manager, and then select the “General - Part 2” tab using EDIT → OBJECT PROPERTIES, you will be provided with the load and work memory requirements for this block.

The length of the user program is listed in the properties of the offline *Blocks* container (select *Blocks* and EDIT → OBJECT PROPERTIES). On the “Blocks” tab you will find the data “Size in work memory” and “Size in load memory”.

Note that the configuration data (system data blocks) are missing in the value for the load memory. The SIMATIC Manager shows you blocks in the detail view with the container open (displayed as table), and the memory requirements in the status line (bottom right in window) with the *System data* object selected.

With the programming device switched online, the current occupation of the CPU memory is shown by the SIMATIC Manager under PLC → DIAGNOSTIC/SETTING → MODULE INFORMATION, “Memory” tab.

Checksum

The program editor generates a checksum for all blocks of the user program, and stores it in the object properties of the *Blocks* container. Identical programs have the same checksum, each change in the program also changes the checksum. A checksum is also generated from the system data. You can view the checksums in the SIMATIC Manager with the *Blocks* container selected and EDIT → OBJECT PROPERTIES.

The checksum of the user program is generated from the program code and the default and initial values of the data blocks. The writing of data addresses in the work memory (actual values) does not change the checksum. The checksum is only adapted when the data blocks are uploaded to the offline data management, when the actual values become the initial values. This also applies to the data blocks generated by a system function.

If a data block generated by system functions is written or deleted, the checksum is not changed. The checksum is adapted if a programmed (loaded) data block is deleted, or if the initial values in the load memory are modified by the system function SFC 84 WRIT_DBL.

3.2 Blocks

You can divide your program into as many sections as you want to in order to make it easier to read and understand. The STEP 7 programming languages support this by providing the necessary functions. Each program section should be self-contained, and should have a technological or functional basis. These program sections are referred to as “Blocks”. A block is a section of a user program which is defined by its function, structure or intended purpose.

3.2.1 Block Types

STEP 7 provides different types of blocks for different tasks:

- ▷ User blocks
Blocks containing user program and user data
- ▷ System blocks
Blocks containing system program and system data
- ▷ Standard blocks
Turnkey, off-the-shelf blocks, such as drivers for FMs and CPs

User blocks

In extensive and complex programs, “structuring” (dividing) of the program into blocks is recommended, and in part necessary. You may choose among different types of blocks, depending on your application:

Organization blocks (OBs)

These blocks serve as the interface between operating system and user program. The CPU’s operating system calls the organization blocks when specific events occur, for example in the event of a hardware or time-of-day interrupt.

The main program is in organization block OB 1. The other organization blocks have permanently assigned numbers based on the events they are called to handle.

Function blocks (FBs)

These blocks are parts of the program whose calls can be programmed via block parameters. They have a variable memory which is located in a data block. This data block is permanently allocated to the function block, or, to be more precise to the function block call. It is even possible to assign a different data block (with the same data structure but containing different values) to each function block call. Such a permanently assigned data block is called an instance data block, and the combination of function block call and instance data block is referred to as a call instance, or “instance” for short. Function blocks can also save their variables in the instance data block of the calling function block; this is referred to as a “local instance”.

Functions (FCs)

Functions are used to program frequently recurring or complex automation functions. They can be parameterized, and return a value (called the function value) to the calling block. The function value is optional, in addition to the function value, functions may also have other output parameters. Functions do not store information, and have no assigned data block.

Data blocks (DBs)

These blocks contain your program's data. By programming the data blocks, you determine in which form the data will be saved (in which block, in what order, and in what data type). There are two ways of using data blocks: as global data blocks and as instance data blocks. A global data block is, so to speak, a “free” data block in the user program, and is not allocated to a code block. An instance data block, however, is assigned to a function block, and stores part of that function block's local data.

The number of blocks per block type and the length of the blocks is CPU-dependent. The number of organization blocks, and their block

Table 3.2 Number ranges for system data blocks

SDB No.	Meaning, Contents
0	CPU parameters which control the response of the operating system and the CPU-internal default settings; overwrites the SDB 2 when transmitting to the CPU
1	Module addressing for central I/O (preset configuration), e.g. assignment of logical and geographical addresses, address space of modules, etc.
2	CPU parameters (default settings in CPU's operating system, become effective following overall reset if a configuration has not yet been transmitted)
3 to 7	Miscellaneous CPU and module parameters, e.g. for saving consistency of transmitted configuration data
20 to 89	Module addressing for distributed I/O (preset configuration), e.g. assignment of logical and geographical addresses, address space of modules, etc.
90 to 99	Configuration data for fault-tolerant and failsafe systems
100 to 149	Parameters for central and distributed modules assigned to the CPU
150 to 152	Parameters for interface submodules
153 to 189	Parameters for distributed modules
200 to 998	Parameters for configuration of communications (e.g. global data communications, symbol-based messages, configuration of connections)
999	Configuration data for routing of connections
1000 and greater	Parameters for distributed I/O, parameters for CP and FM modules, parameters for H/F and TD/OP systems

numbers, are fixed; they are assigned by the CPU's operating system. Within the specified range, you can assign the block numbers of the other block types yourself. You also have the option of assigning every block a name (a symbol) via the symbol table, then referencing each block by the name assigned to it.

System blocks

System blocks are components of the operating system. They can contain programs (system functions (SFCs) or system function blocks (SFBs)) or data (system data blocks (SDBs)). System blocks make a number of important system functions accessible to you, such as manipulating the internal CPU clock, or communications functions.

You can call SFCs and SFBs, but you cannot modify them, nor can you program them yourself. The blocks themselves do not reserve space in user memory; only the block calls and the instance data blocks of the SFBs are in user memory.

SDBs contain information on such things as the configuration of the automation system or the parameterization of the modules. STEP 7 itself generates and manages these blocks. You, however, determine their contents, for instance when you configure the stations. As a rule, SDBs are located in load memory. You cannot open SDBs, and can only read them from your program using special system blocks, e.g. when parameterizing modules.

Double click the *System blocks* object in the *Blocks* container to display a list of current system data blocks generated by the Hardware Configuration tool (in the offline container) or present on the CPU (in the online container). Table 3.2 shows an overview of the numbering system for system data blocks.

Standard blocks

In addition to the functions and function blocks you create yourself, off-the-shelf blocks (called "standard blocks") are also available. They can either be obtained on a storage medium or are

contained in libraries delivered as part of the STEP 7 package (for example IEC functions, or functions for the S5/S7 converter).

Chapter 25 "Block Libraries" contains an overview of the standard blocks supplied in the *Standard Library*.

3.2.2 Block Structure

Essentially, code blocks consist of three parts (Figure 3.2):

- ▷ The block header, which contains the block properties, such as the block name
- ▷ The declaration section, in which the block-local variables are declared, that is, defined
- ▷ The program section, which contains the program and program commentary

A data block is similarly structured:

- ▷ The block header contains the block properties
- ▷ The declaration section contains the definitions of the block-local variables, in this case the data addresses with data type specification
- ▷ The initialization section, in which initial values can be specified for individual data addresses

In incremental programming, the declaration section and the initialization section are combined. You define the data addresses and their data types in the "declaration view", and you can initialize each data address individually in the "data view".

3.2.3 Block Properties

The block properties, or attributes, are contained in the block header. You can view and modify the block properties with the menu command EDIT → OBJECT PROPERTIES in the SIMATIC Manager when the block is selected, or with FILE → PROPERTIES in the Program Editor (Figure 3.3).

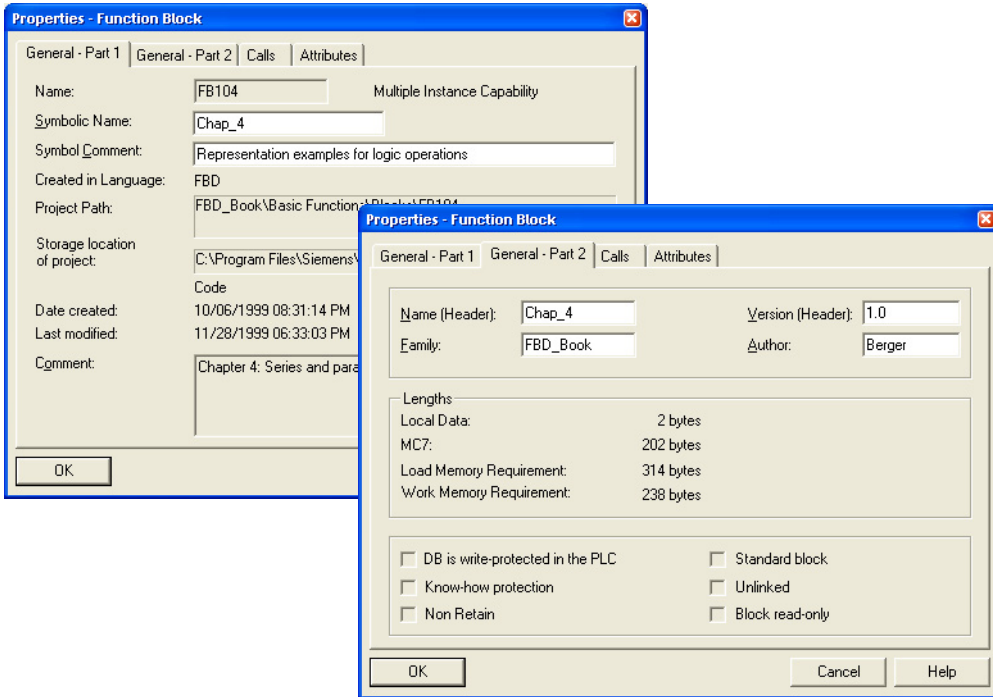


Figure 3.3 Block properties

and the memory locations of the block and the project.

The program editor saves the creation or modification date of the block in two time stamps: these are the block parameters and the static local data for the program code and the interface. Note that the modification date of the interface must be equal to or smaller (older) than the modification date of the program code in the calling block. If this is not the case, the program editor signals a “time stamp conflict” during output of the calling block.

The comment displayed consists of the block title and the block comment which you entered when programming the block.

“General - Part 2” tab

The *Name (Header)* displayed on this tab is the block name; it is not identical to the symbol address. Different blocks can have the same name. Using *Family* you can assign a common feature to a group of blocks. The block name

and family are displayed when inserting blocks if you select the block in the dialog window of the program element catalog. Use *Author* to enter the name of the block's creator. The name, family and author may have up to eight characters each, commencing with a letter. The letters, digits and the underline character are permissible. The *Version* is entered with two 2-digit numbers from 0 to 15.

The length data show the memory allocation for the block in bytes:

- ▷ Local data: allocation in the local data stack (temporary local data)
- ▷ MC 7: size of the block (code only)
- ▷ Load memory requirement
- ▷ Work memory requirement

A block occupies more space in the load memory since the data not relevant to processing are saved here in addition.

The *Know-how protection* attribute is used for block protection. If a block is know-how-pro-

tected, the program in that block can not be viewed, printed out or modified. The Editor shows only the block header and the declaration table with the block parameters. You can assign this attribute during source-oriented input of the block with the keyword `KNOW_HOW_PROTECT`. When you do this to a block, no one can view the compiled version of that block, not even you (make sure you keep the source file in a safe place!).

The attribute *DB is write-protected in the PLC* is an attribute for data blocks only. It means that you can only read that data block in your program. Output of an error message prevents the overwriting of the data in that data block. The write protection applies to the data relevant to processing (actual values) in work memory; the data in load memory (initial values) can be overwritten even if the data block is write-protected. This write protection feature must not be confused with block protection. A data block with block protection can be read out and written to in the user program, but its data can no longer be viewed with a programming or operator monitoring device. The attribute *DB is write-protected in the PLC* is switched off as standard, but can be changed at any time using the program editor. The keyword with source-oriented program input for switching on the write protection is `READ_ONLY`.

The block header of any standard block which comes from Siemens contains the *Standard block* attribute.

Data blocks can be assigned the *Unlinked* attribute. Such data blocks are only present in load memory, and are not relevant to processing. Since their data are not located in work memory, direct access is no longer possible. Data in load memory can be read using system functions and – if the load memory is a micro memory card – also written. Data blocks with the *Unlinked* attribute are suitable for recording data which are only rarely accessed, e.g. recipes or archives. This attribute is switched off as standard, but can be changed at any time using the program editor. The keyword with source-oriented program input for switching on this attribute is `UNLINKED`.

The *Non retain* attribute means “non-retentive” and is assigned to CPUs designed for this for data blocks. If *Non retain* is switched on, the

data block transfers the initial values from load memory to work memory in the event of a power off/on and with a `RUN-STOP` transition (response as with a cold restart). If *Non retain* is switched off, the corresponding data block therefore being retentive, it retains its actual values in the event of a power off/on and with a `RUN-STOP` transition (response as with a warm restart). This attribute is switched off as standard, but can be changed at any time using the program editor. The keyword with source-oriented program input for switching on this attribute is `NON_RETAIN`.

Blocks saved in the Program Editor with the menu command `FILE → STORE READ-ONLY` for reference purposes, for example, receive the block property *Block read-only*. These can be all code blocks, data blocks, and user-defined data types. This property can only be set with the Program Editor, and there is no keyword for source-oriented programming for this purpose.

“Calls” tab

This tab shows a list of all blocks called in this block with the time stamps for the code and the interface. With instance data blocks, the basic function block is shown here together with the local instances (function blocks) called in this instance, in each case with the time stamps for code and interface.

“Attributes” tab

Blocks may have system attributes. System attributes control and coordinate functions between applications, for example in the SIMATIC PCS7 control system.

3.2.4 Block Interface

The declaration table contains the interface of the block to the rest of the program. It consists of the block parameters (input, output and in/out parameters) and also – in the case of function blocks – the static local data. The temporary local data, which do not basically belong to the block interface, are also handled at this point. The block interface is defined in the interface window when programming the block, and is initialized with variables when the

block is called (see Chapter 19 “Block Parameters”).

The Program Editor checks that the block parameter initialization in the called block agrees with the interface of the called block. The Editor uses the time stamp for this: the interface of the called block must be older than the code in the calling block, that is, the last interface modification must have been made prior to integration of the block. The Program Editor updates the interface time stamp when the number of parameters changes or when a data type or a default value changes.

Time stamp conflict

A time stamp conflict occurs when the interface of the called block has a later time stamp than the code of the calling block. You will notice a time stamp conflict if you open an already compiled block again. The Program Editor then indicates the incorrect block call in red. A time stamp conflict can be caused, for example, if you modify the interfaces of blocks that are already called in other blocks, or if you combine blocks from different programs into a new program, or if you re-compile a section of the overall program with a source file.

However, the interface conflict generally described as a “time stamp conflict” can also have other causes. It also occurs if a called or referenced block is younger than the calling block. Examples of the occurrence of time stamp conflicts include the following:

- ▷ The interface of a called block is younger than the code of the calling block.
- ▷ The interface initialization does not agree with the block interface.
- ▷ A function block is younger than its instance data block (the instance data block is generated from the interface description of the function block and should therefore be younger than or the same age as the function block).
- ▷ A local instance is younger than the calling instance (affects function blocks).
- ▷ A user data type UDT is younger than the block whose variables are declared with the UDT; this can be any block including a data block or another UDT.

Correcting invalid block calls

The Program Editor supports you in different manners in finding and correcting invalid block calls. See the next section for how to check the block consistency in a complete program (“Checking block consistency”).

You can check block calls which have become invalid with the block open (with the cursor at the invalid block call) using the menu command **EDIT → BLOCK CALL → UPDATE**. Block calls can become invalid following insertion, deletion or shifting of block parameters, or when changing the name and type.

With **EDIT → BLOCK CALL → CHANGE TO MULTI-INSTANCE CALL** and **EDIT → BLOCK CALL → CHANGE TO FB/DB CALL** you transfer calls from function blocks into local instance calls or into calls with data block. Following modification of the block calls, you must regenerate the associated instance data blocks.

A further possibility is provided by the menu command **FILE → CHECK AND UPDATE ACCESSES**. The invalid block calls in an opened block are then updated or presented for modification.

Checking block consistency

The Program Editor only indicates a time stamp conflict when you open a block containing a time stamp conflict. If you want to check an entire program, you can use the function “Check block consistency” in the SIMATIC Manager. This purges a majority of interface conflicts and directs you to the program locations that require editing.

To carry out a consistency check, select the *Blocks* container in the SIMATIC Manager and then **EDIT → CHECK BLOCK CONSISTENCY**. If a call tree is not displayed in the window, e.g. because the program has been compiled using an earlier version of STEP 7, select **PROGRAM → COMPILE** in this window.

Please note that after checking the block consistency, the instance data blocks and the data blocks generated from the UDT are assigned the *initial* values again in the compiled program.

The Program Editor displays the progress and result of the consistency check in the output

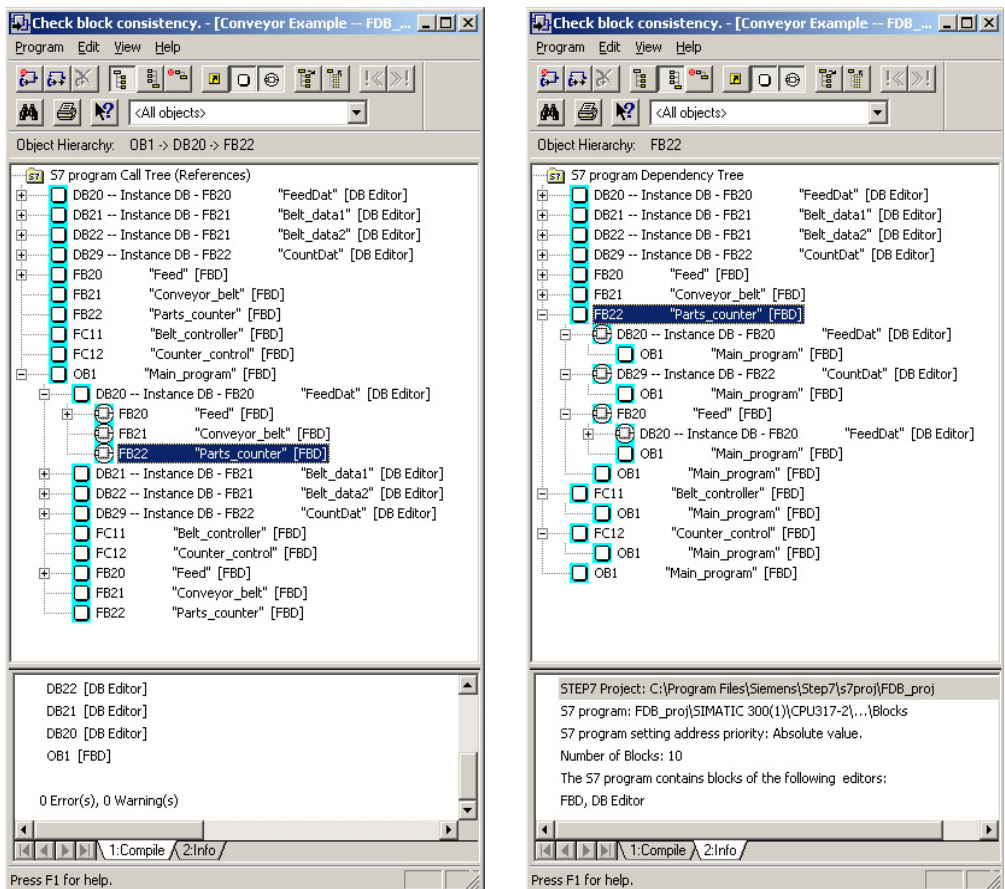


Figure 3.4 Example of the Representation of the “Check Block Consistency” Dependencies

window (VIEW → ERRORS AND WARNINGS). The consistency check cannot be used on programs in libraries.

The dependencies in the case of called or referenced blocks are displayed in the form of a tree diagram (Figure 3.4). You can choose between the following two representations:

The *reference tree* view displays the dependencies in a similar way to the program structure: on the left are the calling blocks, further to the right are the blocks called in the blocks on the left. Example: instance DB 20/FB 20 is called in OB 1 and local instances FB 21 and FB 22 are called in FB 20.

The *dependency tree* view displays the dependencies starting from all called or referenced

blocks. They are located in the left-hand column, and the calling blocks are listed to the right of this. Example: FB 22 stores its data in instance DB 20/FB 20 that is called in OB 1. It also has its own DB 29 and it is called as a local instance in FB 20.

The determined information is displayed in compact form by symbols. An exclamation mark, for example, indicates that the object caused a time stamp conflict. A white cross on a red background indicates that the associated block must be recompiled.

If you select a block in the tree diagram or in the output window, you can edit it with EDIT → OPEN OBJECT, e.g. correct an incorrect call.

3.3 Programming Code Blocks

Chapter 2.5 “Creating the S7 Program” contains an introduction to program creation and to operating the program editor.

3.3.1 Opening Blocks

You begin block programming by opening a block. Open an existing block either by double-clicking on the block in the SIMATIC Manager's project window or by selecting **FILE** → **OPEN** in the program editor.

If you open a compiled block in the *Blocks* container, e.g. by double-clicking, it is open for incremental programming. This is the case both with offline and online programming.

If the block does not yet exist, you can generate it in the following ways:

- ▷ In the SIMATIC Manager: Select the *Blocks* object in the left-hand part of the project window and create a new data block with **INSERT** → **S7 BLOCK** → **.....**. You are provided with the properties window of the block. On the “General – Part 1” tab, select the number of the block under *Name* and the language “LAD” or “FBD”. You can enter the remaining attributes later.
- ▷ In the Editor with menu command **FILE** → **NEW**, which displays a dialog box in which you can enter the desired block under *object name*. After closing the dialog box you can program the contents of the block. The Program Editor uses the language set on the “Create Block” tab under **OPTIONS** → **CUSTOMIZE**.

You can enter the information for the block header when you generate the block or you can

enter the block attributes later in the Editor by opening the block and selecting the menu command **FILE** → **PROPERTIES**.

3.3.2 Block Window

The program editor shows the variable declaration table (block parameters and local data) and the program window (code and comments) for an opened code block. The program elements cannot be additionally displayed in the overview window (Figure 3.5).

Variable declaration table

The variable declaration table is in the window above the program window. If it is not visible, position the mouse pointer to the upper line of demarcation for the program window, click on the left mouse button when the mouse pointer changes its form, and pull down. You will see the overview of the types of variable on the left, and the variable declaration table on the right, which is where you define the block-local variables (see Table 3.3).

In order to declare a variable, select its type in the left area, and fill in the table on the right. Not every type of variable can be programmed in every kind of code block. If you do not use a variable type, the corresponding line remains empty.

The declaration for a variable consists of the name, the data type, a default value, if any, and a variable comment (optional). Not all variables can be assigned a default value (for instance, it is not possible for temporary local data). The default values for functions and function blocks are described in detail in Chapter 19 “Block Parameters”.

Table 3.3 Variable Types in the Declaration Section

Variable Type	Declaration	Possible in Block Type		
Input parameters	IN	-	FC	FB
Output parameters	OUT	-	FC	FB
In-out parameters	IN_OUT	-	FC	FB
Static local data	STAT	-	-	FB
Temporary local data	TEMP	OB	FC	FB
Function value	RETURN	-	FC	-

can drag all program elements into the program window using the mouse.

In addition, it lists the blocks already located in the offline *Blocks* container, as well as the already-programmed multi-instances and the available libraries. By clicking with the right mouse button on a block or a block type, you can select whether the blocks are to be sorted according to type and number or according to the block family.

Call structure

The call structure shows the block hierarchy in the current user program. You are shown the call environment of the currently opened block together with the blocks used.

3.3.4 Programming Networks

You can divide a LAD/FBD program into networks which each represent a current path or a logic operation. The Editor numbers the networks automatically, beginning with 1. Each block can accommodate up to 999 networks. You may give each network a network title and a network comment. During editing, you can select each network directly with the menu command **EDIT → GO TO → ...**.

To enter the program code, click once below the window for the network comment, or, if you have set “Display with Comments”, click once below the shaded area for network comments. You will see a framed empty window. You can begin entering your program anywhere within this window. The chapters below show you what a LAD current path or an FBD logic operation looks like.

You program a new network with **INSERT → NETWORK**. The Editor then inserts an empty network behind the currently selected network.

You need not terminate a block with a special statement, simply stop making entries. However, you can program a last (empty) network with the title “Block End”, providing an easily seen visual end of the block (an advantage, particularly in the case of exceptionally long blocks).

In the Program Editor, you can create new blocks, or open and edit existing blocks with-

out having to change back to the SIMATIC Manager.

Network templates

Just as you store blocks in a library to reuse them in other programs, you also save network templates in order to copy them again and again in, for example, other blocks.

To save network templates, create a library containing at least one S7 program and the *Sources* container.

You program the networks that you want to use as templates quite “normally” in (any) block. Then you replace the addresses that are to change with the dummy characters %00 to %99. You can also vary the network title and the network comment in this way.

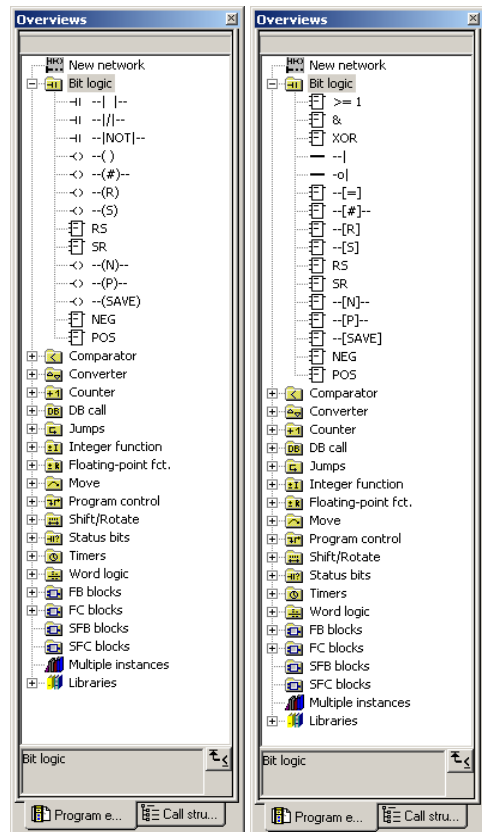


Figure 3.6
Program Elements Catalog for LAD and FBD

The dummy characters replacing the addresses are presented in red because a block cannot be stored in this form. This is not significant because following saving of the network template(s), this block can be rejected (close the block without saving).

After entering the dummy characters, mark the network by clicking on the network number at top left before the network title. You can also combine several networks to form one template; hold down the Ctrl key while you click on further network numbers.

Now select EDIT → CREATE NETWORK TEMPLATE. In the dialog box that then appears, you can assign meaningful comments to the network and all the dummy characters. In the next dialog box, you assign a name for the network template and you define the storage location (*Sources* container in a library).

If you want to use the network templates, open the relevant library in the Program Elements Catalog and then select the desired network template (double-click or drag to the Editor window). A dialog box appears automatically and here you replace the dummy characters with valid entries. The network template is inserted after the selected network.

3.3.5 Addressing

The addresses used in the program, such as inputs and outputs or bit memories, are addressed in absolute or symbolic mode.

Absolute addressing

Absolute addressing references addresses and block parameters with the address ID and the bit/byte address. If there are three red question marks in the network in the place of addresses and parameters, you must replace this character string with valid addresses. If there are three black points, replacement is optional.

The Program Editor checks that the data types of the addresses and parameters are correct. You can deactivate some of these type checks (in the Program Editor under OPTIONS → CUSTOMIZE, “LAD/FBD” tab, “Type check for addresses” option).

Symbolic addressing

If you want to use symbolic names for global operands in incremental programming, these names must already be assigned to absolute addresses in the Symbol Table. While entering the program with the Program Editor, you can call up the Symbol Table for editing with OPTIONS → SYMBOL TABLE and then you can change symbols or enter new symbols.

You activate display of the symbol addresses with VIEW → DISPLAY WITH → SYMBOLIC REPRESENTATION. The menu point VIEW → DISPLAY WITH → SYMBOL INFORMATION provides, for each network, a list of the symbol-to-absolute-address assignments for each symbol used in the network.

While entering the symbols, you can view a list of all the symbols in the symbol table with INSERT → SYMBOL (or right mouse click and INSERT SYMBOL) and you can then transfer one of the symbols with a click of the mouse. The list is displayed automatically if you have set VIEW → DISPLAY WITH → SYMBOL SELECTION.

If a symbol is not yet included in the symbol table, you can select EDIT → SYMBOLS, make the assignment to the absolute address, and possibly also assign a symbol comment. This symbol is then transferred to the symbol table when you click OK.

You can also edit the symbols in Register “4: Address info” in the Details window. If the columns with the symbol and the symbol comment are not displayed, fetch them by clicking with the right mouse button on the address table and DISPLAY COLUMNS (ON/OFF).

If the Program Editor opens a compiled block, it carries out “decompiling” to the LAD or FBD method of representation. In doing so, it uses the non-execution-relevant program sections in the offline data management, in order, for example, to represent symbols, comments and jump labels. If the information from the offline data management system is missing, the Program Editor uses substitute symbols.

3.3.6 Editing LAD Elements

Programming in general

The program consists of individual LAD elements arranged in series or parallel to one another. Programming of a current path, or rung, begins on the left power rail. You select the location in the rung at which you want to insert an element, then you select the program elements you want

- ▷ with the corresponding function key (for example F2 for a normally open (NO) contact),
- ▷ with the corresponding button on the function bar or
- ▷ from the Program Elements Catalog (with INSERT → PROGRAM ELEMENTS or VIEW → OVERVIEWS).

You terminate a rung with a coil or a box.

Most program elements must be assigned memory locations (variables). The easiest way to do this is to first arrange all program elements, and then label them.

Contacts

Binary addresses such as inputs are scanned using contacts. The scanned signal states are combined according to the arrangement of the contacts in a serial or parallel layout.

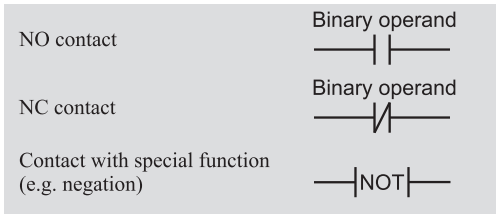
“Current flows” through a *normally open contact* if the scanned binary address has signal state “1” (the contact is activated) “current flows” through a *normally closed contact* if the scanned binary address has signal state “0” (the contact is not activated). You can also scan status bits or negate the result of the logic operation (NOT contact).

Coils

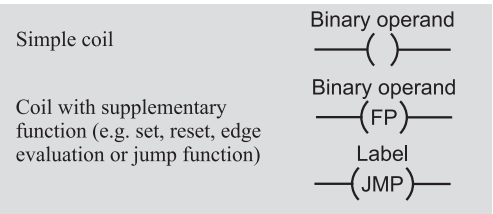
Coils are used to control binary addresses, such as outputs. A simple coil sets the binary address when current flows into the coil, and resets it when current no longer flows.

There are coils with additional labels, such as Set and Reset coils, which serve a special function. You can also use coils to control timers and counters, call blocks without parameters, execute jumps in the program, and so on.

Contacts



Coils



Boxes

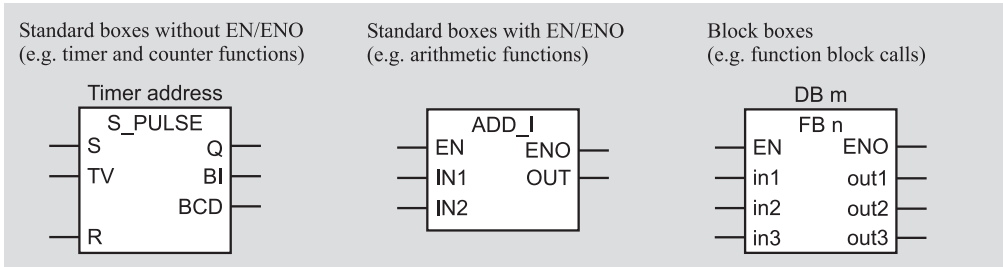


Figure 3.7 Examples of LAD Program Elements

Network 2: Parts ready to remove

When the parts have reached the end of the belt, they are ready for removal.

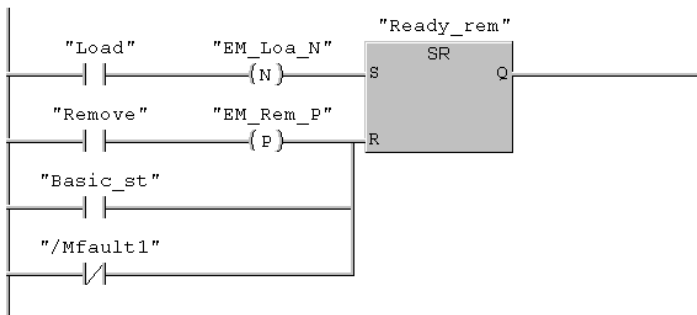


Figure 3.8 Example of a LAD Network in “3-Dimensional” Representation

Boxes

Boxes represent LAD elements with complex functions. STEP 7 provides “standard boxes” of two different types: without EN/ENO mechanism (such as memory functions, timer and counter functions, comparison boxes), and with EN/ENO (such as MOVE, arithmetic and math functions, data type conversions). When you call code blocks (FCs, FBs, SFCs and SFBs), LAD also represents the calls as boxes with EN/ENO. LAD also provides an “empty box” in which you can enter the desired function when programming.

Layout restrictions

The LAD editor sets up a network according to the “main rung” principle. This is the uppermost branch, which begins directly on the left power rail and must terminate with a coil or a box. All LAD elements can be located in this rung. In parallel branches which do not begin on the left power rail, there are sometimes restrictions in the choice of program elements.

Additional restrictions dictate that no LAD element may be “short-circuited” with an “empty” parallel branch, and that no “power” may flow through an element from right to left (a parallel branch must be closed to the branch in which it was opened). Any further rules applying to the layout of special LAD elements are discussed in the relevant chapters.

When using boxes as program elements, you can

- ▷ program a single box per network
- ▷ arrange boxes in T branches in branches that start at the left power rail
- ▷ arrange boxes in series by switching the ENO output of one box to the EN input of the following box
- ▷ switch boxes in parallel in branches on the left power rail via its ENO output

With the arrangement of the boxes, you evaluate the signal states of the ENO outputs: if you terminate the ENO outputs with a coil, “power” flows into the coil if all the boxes have all been processed without errors in the case of series connection, or if one of the boxes has been processed without errors in the case of parallel connection (see also Chapter 15.4 “Using the Binary Result”).

3.3.7 Editing FBD Elements

Programming in general

The program consists of individual program elements that are connected together via the binary signal flow to form logic operations or networks. You begin programming a logic operation by selecting the programming elements on the left of the logic operation

- ▷ with the function key,
(e.g. F2 for the AND function),
- ▷ via the menu
(INSERT → FBD LANGUAGE ELEMENTS → AND BOX) or
- ▷ from the Program Elements Catalog
(with INSERT → PROGRAM ELEMENTS or VIEW → OVERVIEWS).

You terminate a binary logic operation in the simplest case with an assign box.

Most program elements must be assigned memory locations (variables). The easiest way to do this is to first arrange all program elements, then label them.

Binary functions

You scan the binary addresses such as inputs and combine the scanned signal states using the binary functions AND, OR and exclusive OR. Each binary input of a box also scans the binary address at the input.

The scanning of an address can be negated so that scan result “1” can be obtained for status

“0” of the address. You can also scan status bits and negate the result of the logic operation.

Simple boxes

You control binary addresses such as outputs with simple boxes. Simple boxes generally have only one input and may have an additional label

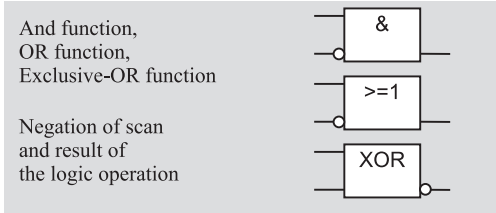
There are simple boxes for controlling a binary address, evaluating an edge, controlling timer and counter addresses, calling blocks without parameters, executing jumps in the program, and so on.

Complex boxes

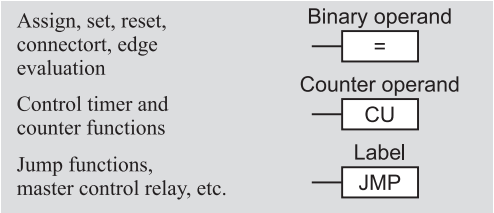
Complex boxes represent program elements with complex functions. STEP 7 provides “standard boxes” in two versions:

- ▷ without EN/ENO mechanism (such as memory functions, timers and counters, comparison boxes) and
- ▷ with EN/ENO (such as MOVE, arithmetic and math functions, data type conversion).

Binary functions



Simple boxes



Complex boxes

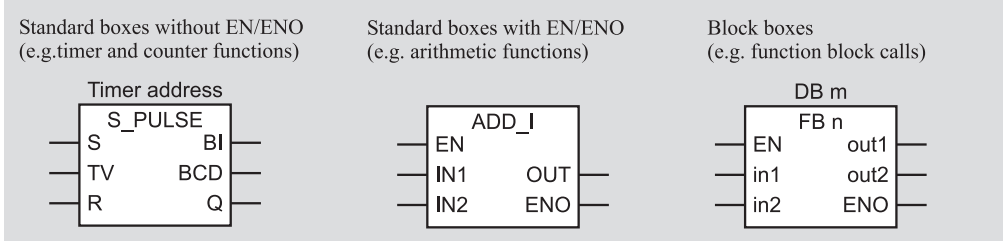
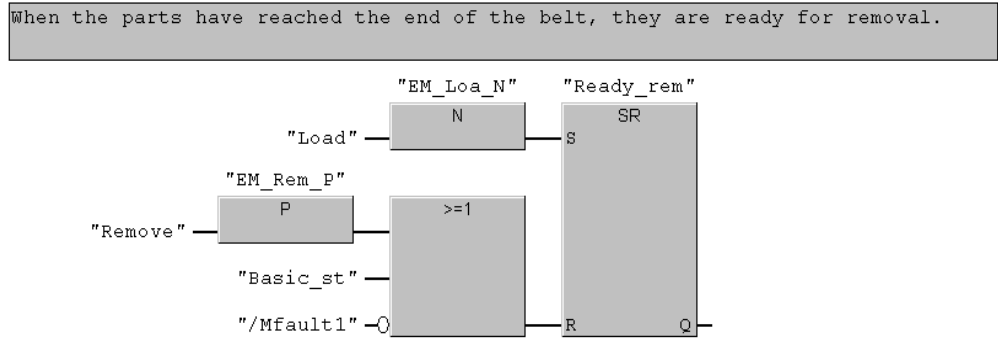


Figure 3.9 Examples of FBD Program Elements

Network 2: Parts ready to remove**Figure 3.10** Example of an FBD Network in “3-Dimensional” Representation

If you call code blocks (FCs, FBs, SFCs and SFBs), FBD also represents the calls as boxes with EN/ENO.

FBD also provides an “empty box” in which you can enter the desired function when programming.

Layout restrictions

The FBD editor sets up a network from left to right and from the top down. From the left, the inputs lead to the functions and the outputs exit to the right.

A logic operation always has a “terminating function”. In its simplest form, this is an assignment of the result of the logic operation to a binary address.

With the help of a T branch of a logic operation, you can program further “terminating functions” for a logic operation (“multiple output”). Following a T branch however, the selection of programmable elements is restricted. For example, you cannot arrange edge evaluations and call boxes following a T branch. Any further rules applying to the layout of special FBD elements are discussed in the relevant chapters.

When using boxes as program elements, you can

- ▷ program a single box per network
- ▷ arrange boxes in T branches in branches that start at the left power rail

- ▷ arrange boxes in series by switching the ENO output of one box to the EN input of the following box
- ▷ AND or OR boxes via the ENO output.

In the case of boxes switched in series, you can control their processing as a group (see also Chapter 15.4 “Using the Binary Result”). You evaluate the error messages of the boxes by combining the ENO outputs: ANDing of the ENO outputs is fulfilled if all boxes have been processed without error, and ORing of the ENO outputs is fulfilled if one of the boxes has been processed without error.

3.4 Programming Data Blocks

Chapter 2.5 “Creating the S7 Program” gives an introduction to program creation and the use of the program editor. Data blocks are programmed in the same way in LAD and FBD.

3.4.1 Creating Data Blocks

You begin block programming by opening a block, either with a double-click on the block in the project window of the SIMATIC Manager or by selecting FILE → OPEN in the editor. If the block does not yet exist, create it as follows:

- ▷ In the SIMATIC Manager: select the object *Blocks* in the left-hand portion of the project window and create a new data block with INSERT → S7 BLOCK → DATA BLOCK. You

see the properties window of the block. Specify the number and type of the data block on the “General – Part 1” tab (see Chapter 3.4.2 “Types of Data Blocks”). “Instance DB” and “DB of type” can only be selected if function blocks FB, system function blocks SFB or user data types UDT are present in the block container. You can also enter the remaining block properties later.

- ▷ In the program editor: with FILE → NEW, you get a dialog box in which you can enter the desired block under “Object name”. In the dialog window “New data block” which is subsequently displayed, you are requested to define the type of data block (see Chapter 3.4.2 “Types of Data Blocks”). After closing the dialog box, you can program the block contents.

You can fill out the header of a block as you create it or you can add the block properties at a later point. You program later additions to the block header in the editor by selecting FILE → PROPERTIES while the block is open.

3.4.2 Types of Data Blocks

When creating a new data block, you are requested to define its type. When creating using the SIMATIC Manager, you set the type in the selection box of the properties window; when creating with the program editor, by clicking one of the options offered in the “New data block” window.

A differentiation is made between three types of data block depending on their creation and application:

- ▷ “Data block” or “Shared DB”

Creation as a global data block; you declare the data addresses when programming the data block in this case

- ▷ “Data block referencing a user-defined data type” or “DB of type”

Creation as a data block of user data type; in this case the data structure is used which you have declared when programming the corresponding user data type UDT.

- ▷ “Data block referencing a function block” or “Instance DB”

Creation as an instance data block; here, the data structure that you have declared when programming the relevant function block is transferred.

When creating a data block on the basis of a user data type, you simultaneously define the UDT on which it is based; i.e. the UDT must already have been present in the block container. The same applies to the creation of a data block with assigned function block.

3.4.3 Block Windows and Views

When opening a data block whose structure is based on a user data type or a (system) function block, you will be asked in the standard setting whether you wish to open the data block using the program editor or the application “Parameterization of data blocks”. The parameter view presents the data values grouped technologically, and permits more convenient parameterization (see Chapter 2.7.8 “Monitoring and Modifying Data Addresses”). The data views are described below.

The program editor provides two views for programming (creating) data blocks:

- ▷ The declaration view is used to define the data structure for global data blocks, as well as the default values.
- ▷ You can handle the online values in the data view.

Each view presents a table containing the absolute data addresses in sequence, the names and data types, the initial values and comments (Figure 3.11). The data view contains an additional column with the actual value.

If you open a data block from the offline data management, you are provided with the offline window with which you can edit the data in the programming device. If you open a data block which is present in the CPU’s user memory, the editor displays the online window with which you can edit the data values on the CPU.

Offline window

You use the declaration view for inputs with global data blocks. You declare the data addresses in this view: you define the sequence

Address	Name	Type	Initial Value	Comment
0.0		STRUCT		
+0.0	Number_1	INT	0	
+2.0	Number_2	INT	0	
+4.0	Sum	INT	0	
+6.0	FirstVol	STRUCT		Example of a STRUCT variable
+0.0	FirstWid	INT	0	
+2.0	FirstLen	INT	0	
+4.0	FirstHei	INT	0	
+6.0	FirstTime	TIME_OF_DAY	TOD#0:0:0.0	
=10.0		END_STRUCT		
+16.0	Meas1	STRUCT		Example of nested STRUCT variable
+0.0	MeasTime	TIME	T#0MS	
+4.0	Volume1	STRUCT		
+0.0	Width1	INT	0	
+2.0	Length1	INT	0	
+4.0	Height1	INT	0	
+6.0	MeasTime1	TIME_OF_DAY	TOD#0:0:0.0	
=10.0		END_STRUCT		
=14.0		END_STRUCT		
+30.0	First_name	STRING[8]	'Hans'	Example of a STRING variable
+40.0	Last_name	STRING[14]	'Berger'	Example of a STRING variable
+56.0	Header_data	"Header"		Use of UDT 101 "Header"
=64.0		END_STRUCT		

Figure 3.11 Example of an Opened Data Block (Declaration View)

of data addresses, assign a name and data type to each data address, and can additionally enter a comment. Each data address is assigned a default value. This is zero, the smallest value or empty depending on the data type. You can modify the default value in the initial value column.

The data addresses and the default values are already defined for data blocks which are derived from a user data type or from a function block. They are obtained from the declaration of the user data type or from the declaration of the function block.

The data view additionally shows the actual value column. The default values from the initial value column are entered as standard in this column. In the data view, you can enter a different initial value for the load memory and thus an actual value for the work memory (Figure 3.12).

The possibility which exists for assigning individual default values to each data block is particularly important for the data blocks derived from a user data type or from a function block. For example, if you generate several instance data blocks of a function block, all data blocks have the default value set in the function block. In the data view, you can now individually assign other values to various data addresses for each instance.

Online window

You usually use the online window to view the actual data values in the user memory. However, you can also use it to generate data blocks.

The initial value column in the declaration view shows the initial value from the offline data management or the initial value from the load memory if the offline project associated with

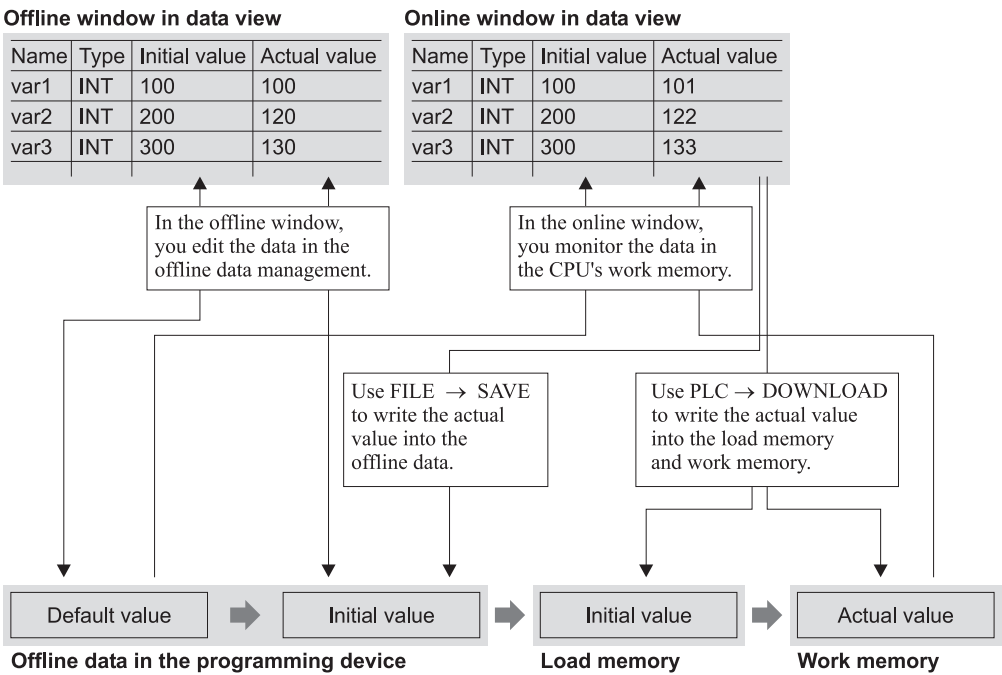


Figure 3.12 Data Storage with Incremental Programming

the CPU program is not available. The actual value column in the data view displays the actual value from the work memory. With EDIT → INITIALIZE DATA BLOCK you can request the editor to replace all actual values by the initial values again.

When writing back with PLC → DOWNLOAD, you write the value in the actual value column into the work memory. You are therefore able to use the programming device to influence the values of data addresses during program execution. The value in the initial value column is rejected.

When writing back with FILE → SAVE, you write the value in the initial value column as the default value, and the value in the actual value column as the initial value into the offline data management.

Note that the complete information concerning data addresses, such as e.g. the name, is only present in the offline data management. It is recommendable to also write the data blocks generated in the CPU's user memory into the offline data management so that data consistency is retained (Chapter 2.6.5 "Block Handling" under "Data blocks offline/online").

3.5 Variables, Constants and Data Types

3.5.1 General Remarks Concerning Variables

A variable is a value with a specific format (Figure 3.13). Simple variables consist of an address (such as input 5.2) and a data type (such as BOOL for a binary value). The address, in turn, comprises an address identifier (such as I for input) and an absolute storage location (such as 5.2 for byte 5, bit 2). You can also reference an address or a variable symbolically by assigning the address a name (a symbol) in the symbol table.

A bit of data type BOOL is referred to as a *binary address* (or *binary operand*). Addresses comprising one, two or four bytes or variables

with the relevant data types are called *digital operands*.

Variables, which you declare within a block, are referred to as (block-) local variables. These include the block parameters, the static and temporary local data, even the data addresses in global data blocks. When these variables are of an elementary data type, they can also be accessed as operands (for instance static local data as DI operands, temporary local data as L operands, and data in global data blocks as DB operands).

Local variables, however, can also be of complex data type (such as structures or arrays). Variables with these data types require more than 32 bits, so that they can no longer, for example, be loaded into the accumulator. And for the same reason, they cannot be addressed with “normal” STL statements. There are special functions for handling these variables, such as the IEC functions, which are provided as a standard library with STEP 7 (you can generate variables of complex data type in block parameters of the same data type).

If variables of complex data type contain components of elementary data type, these components can be treated as though they were separate variables (for example, you can load a component of an array consisting of 30 INT values into the accumulator and further process it).

Constants are used to preset variables to a fixed value. The constant is given a specific prefix depending on the data type.

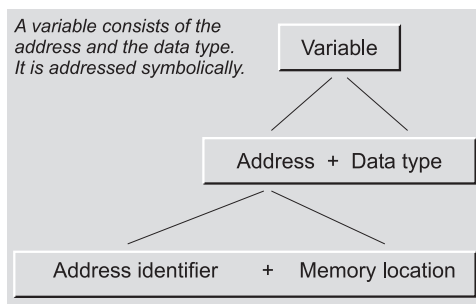


Figure 3.13 Structure of a Variable

3.5.2 Addressing Variables

When addressing variables, you may choose between absolute addressing and symbolic addressing.

- ▷ Absolute addressing uses numerical addresses beginning with zero for each address area.
- ▷ Symbolic addressing uses alphanumeric names, which you yourself define in the symbol table for global addresses or in the declaration section for block-local addresses.

Absolute addressing of variables

Variables of elementary data type can be referenced by absolute addresses.

The absolute address of an input or output is computed from the module start address, which you set or had set in the configuration table and the type of signal connection on the module. A distinction is made between binary signals and analog signals.

Binary signals

A binary signal contains one bit of information. Examples of binary signals are the input signals from limit switches, momentary-contact switches and the like which lead to digital input modules, and output signals which control lamps, contactors, and the like via digital output modules.

Analog signals

An analog signal contains 16 bits of information. An analog signal corresponds to a “channel”, which is mapped in the controller as a word (2 bytes) (see below). Analog input signals (such as voltages from resistance thermometers) are carried to analog input modules, digitized, and made available to the controller as 16 information bits. Conversely, 16 bits of information can control an indicator via an analog output module, where the information is converted into an analog value (such as a current).

The information width of a signal also corresponds to the information width of the variable in which the signal is stored and processed. The information width and the interpretation of the information (for instance the positional

weight), taken together, produce the data type of the variable. Binary signals are stored in variables of data type BOOL, analog signals in variables of data type INT.

The only determining factor for the addressing of variables is the information width. In STEP 7, there are four widths, which can be accessed with absolute addressing:

- ▷ 1 bit Data type BOOL
- ▷ 8 bits Data type BYTE or another data type with 8 bits
- ▷ 16 bits Data type WORD or another data type with 16 bits
- ▷ 32 bits Data type DWORD or another data type with 32 bits

Variables of data type BOOL are referenced via an address identifier, a byte number, and – separated by a decimal point – a bit number. Numbering of the bytes begins at zero for each address area. The upper limit is CPU-specific. The bits are numbered from 0 to 7.

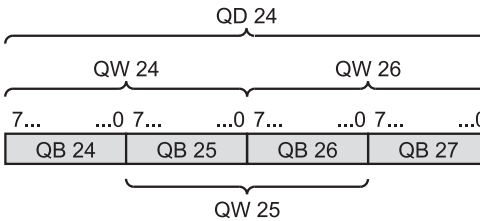


Figure 3.14
Byte Contents in Words and Doublewords

Examples:

- I 1.0 Input bit no. 0 in byte no. 1
- Q 16.4 Output bit no. 4 in byte no. 16

Variables of data type BYTE have as absolute address the address identifier and the number of the byte containing the variable. The address identifier is supplemented by a B. Examples:

- IB 2 Input byte no. 2
- QB 18 Output byte no. 18

Variables of data type WORD consist of two bytes (a word). They have as absolute address the address identifier and the number of the

low-order byte of the word containing the variable. The address identifier is supplemented by a W. Examples:

- IW 4 Input word no. 4;
 contains bytes 4 and 5
- QW 20 Output word no. 20;
 contains bytes 20 and 21

Variables of data type DWORD consist of four bytes (a doubleword). They have as absolute address the address identifier and the number of the low-order byte of the word containing the variable. The address identifier is supplemented by a D. Examples:

- ID 8 Input doubleword no. 8;
 contains bytes 8, 9, 10 and 11
- QD 24 Output doubleword no. 24;
 contains bytes no. 24, 25, 26 and 27

Addresses for the data area include the data block. Examples:

- DB 10.DBX 2.0
 Data bit 2.0 in data block DB 10
- DB 11.DBB 14
 Data byte 14 in data block DB 11
- DB 20.DBW 20
 Data word 20 in data block DB 20
- DB 22.DBD 10
 Data doubleword 10 in data block DB 22

Additional information on addressing the data area can be found in Chapter 18.2.2 “Accessing Data Operands”.

Symbolic addressing of variables

Symbolic addressing uses a name (called a symbol) in place of an absolute address. You yourself choose this name. Such a name must begin with a letter and may comprise up to 24 characters. A keyword is not permissible as a symbol.

There is no difference between upper-case and lower-case letters when entering a symbol. During the output, the editor applies the notation defined during declaration of the symbol.

The name, or symbol, must be allocated to an absolute address. A distinction is made

between global symbols and symbols that are local to a block.

Global symbols

You may assign names in the symbol table to the following objects:

- ▷ Data blocks and code blocks
- ▷ Inputs, outputs, peripheral inputs and peripheral outputs
- ▷ Memory bits, timers and counters
- ▷ User data types
- ▷ Variable tables

A global symbol may also include spaces, special characters and country-specific characters such as the umlaut. Exceptions to this rule are the characters 00_{hex} and FF_{hex} and the quotation mark ("). When using symbols containing special characters, you must put the symbols in quotation marks in the program. In compiled blocks, the STL Editor always shows global symbols in quotation marks.

You can use global symbols throughout the program; each such symbol must be unique within a program.

Editing, importing and exporting of global symbols are described in Chapter 2.5.2 “Symbol Table”.

Block-local symbols

The names for the local data are specified in the declaration section of the relevant block. These names may contain only letters, digits and the underline character (no umlauts!).

Local symbols are valid only within a block. The same symbol (the same variable name) may be used in a different context in another block. The Editor shows local symbols with a leading “#”. When the Editor cannot distinguish a local symbol from an address, you must precede the symbol with a “#” character during input.

Local symbols are available only in the programming device database (in the offline container *Blocks*). If this information is missing on decompilation, the Editor inserts a substitute symbol.

Using symbol names

If you use symbolic names while programming with the incremental Editor, they must have already been allocated to absolute addresses. You also have the option of entering new symbolic names in the symbol table during program input. You can subsequently continue program input with the new symbolic name.

If you compile a source text file generated e.g. from LAD/FBD blocks, the complete assignment of symbolic names to absolute addresses is only made available during the compilation.

In the case of arrays, the individual components are accessed via the array name and a subscript, for example MSERIES[1] for the first component. In LAD and FBD, the index is a constant INT value.

In structures, each subidentifier is separated from the preceding subidentifier by a decimal point, for instance FRAME.HEADER.CNUM. Components of user data types are addressed exactly like structures.

Data addresses

Symbolic addressing of data uses complete addressing including the data block. Example: the data block with the symbolic address MVALUES contains the variables MVALUE1, MVALUE2 and MTIME. These variables can be addressed as follows:

```
"MVALUES".MVALUE_1
"MVALUES".MVALUE_2
"MVALUES".MTIME
```

Please refer to Chapter 18.2.2 “Accessing Data Operands” for further information on addressing of data.

3.5.3 Overview of Data Types

Data types stipulate the characteristics of data, essentially the representation of the contents of a variable, and the permissible ranges. STEP 7 provides predefined data types, which you can combine into user data types.

The data types are available on a global basis, and can be used in every block. LAD and FBD use the same data types.

Table 3.4 Division of the Data Types

Elementary Data Types	Complex Data Types	User Data Types	Parameter Data Types
BOOL, BYTE, CHAR, WORD, INT, DATE, DWORD, DINT, REAL, S5TIME, TIME, TOD	DT, STRING, ARRAY, STRUCT	UDT, Global data blocks, Instances	TIMER, COUNTER, BLOCK_DB, BLOCK_SDB, BLOCK_FC, BLOCK_FB, POINTER, ANY
Data types comprising no more than one doubleword (32 bits)	Data types that can comprise more than one doubleword (DT, STRING) or which consist of several components	Structures or data areas which can be assigned a name	Block parameters
Can be mapped to operands referenced with absolute and symbolic addressing	Can be mapped only to variables that are addressed symbolically		Can be mapped only to block parameters (symbolic addressing only)
Permitted in all address areas	Permitted in data blocks (as global data and instance data), as temporary local data and as block parameters		Permitted in conjunction with block parameters

Depending on structure and application, the data types with STEP 7 are classified as follows:

- ▷ Elementary data types
- ▷ Complex data types
- ▷ User data types
- ▷ Parameter types

Table 3.4 shows the properties of these data type classes.

You can find examples of the declaration and use of variables of all data types in the libraries “LAD_Book” and “FBD_Book” under the program “Data Types” program that you can download from the publisher's Website (see page 8).

3.5.4 Elementary Data Types

Elementary data types can reserve a bit, a byte, a word or a doubleword.

Table 3.6 shows the elementary data types. For many data types, there are two constant representations that you can use equally (e.g. TIME# or T#). The table contains the minimum value for a data type in the upper line and the maximum value in the lower line.

Declaration of elementary data types

Table 3.5 shows some examples of the declaration of variables of elementary data types. *Name* is the identifier for a block-local variable (up to 24 characters, alphanumeric and underscore only). You enter the associated data type in the *Type* column.

Table 3.5 Examples of Declaration and Initial Value for Elementary Data Types

Name	Type:	Initial Value	Comments
Automatic	BOOL	FALSE	Initial value is signal state “0”
Manual_off	BOOL	TRUE	Initial value is signal state “1”
Measured_value	DINT	L#0	Initial value of a DINT variable
Memory	WORD	W#16#FFFF	Initial value of a WORD variable
Waiting_time	S5TIME	S5T#20s	Initial value of an S5 time variable

Table 3.6 Overview of Elementary Data Types

Data Type	(Width)	Description	Example for Constant Notation
BOOL	(1 bit)	Bit	FALSE TRUE
BYTE	(8 bits)	8-bit hexadecimal number	B#16#00, 16#00 B#16#FF, 16#FF
CHAR	(8 bits)	One character (ASCII)	Printable character, e.g. 'A'
WORD	(16 bits)	16-bit hexadecimal number	W#16#0000, 16#0000 W#16#FFFF, 16#FFFF
		16-bit binary number	2#0000_0000_0000_0000 2#1111_1111_1111_1111
		Count value, 3 decades BCD	C#000 C#999
		Two 8-bit unsigned decimal numbers	B#(0,0) B#(255,255)
DWORD	(32 bits)	32-bit hexadecimal number	DW#16#0000_0000, 16#0000_0000 DW#16#FFFF_FFFF, 16#FFFF_FFFF
		32-bit binary number	2#0000_0000_..._0000_0000 2#1111_1111_..._1111_1111
		Four 8-bit unsigned decimal numbers	B#(0,0,0,0) B#(255,255,255,255)
INT	(16 bits)	Fixed-point number	-32 768 +32 767
DINT	(32 bits)	Fixed-point number	L#-2 147 483 648 ¹⁾ L#+2 147 483 647 ¹⁾
REAL	(32 bits)	Floating-point number	+1.234567E+02 ²⁾ in exponential representation
			123.4567 ²⁾ as decimal number
S5TIME	(16 bits)	Time value in SIMATIC format	S5T#0ms S5TIME#2h46m30s
TIME	(32 bits)	Time value in IEC format	T#-24d20h31m23s647ms TIME#24d20h31m23s647ms
			T#-24.855134d TIME#24.855134d
DATE	(16 bits)	Date	D#1990-01-01 DATE#2168-12-31
TIME_OF_DAY	(32 bits)	Time of day	TOD#00:00:00.000 TIME_OF_DAY#23:59:59.999

¹⁾ "L#" may be omitted if the number is outside the INT number range²⁾ for value range see text

With the exception of the temporary local data and block parameters of functions, you can assign an *initial value* to the variables. Use the syntax suitable for the data type for this purpose. *Comments* are optional.

BOOL, BYTE, WORD, DWORD, CHAR

A variable of data type BOOL represents a bit value (for example input I 1.0). Variables with data types BYTE, WORD and DWORD are bit strings comprising 8, 16 and 32 bits, respectively. The individual bits are not evaluated.

Special forms of these data types are the BCD numbers and the count as used in conjunction with counter functions, as well as data type CHAR, which represents an ASCII character.

BCD numbers

BCD numbers have no special identifier. Simply enter a BCD number with the data type 16# (hexadecimal) and use only digits 0 to 9.

BCD numbers occur in coded processing of time values and counts and in conjunction with conversion functions. Data type S5TIME# is used to specify a time value for starting a timer (see below), data type 16# or C# for specifying a count value. A C# count value is a BCD number between 000 and 999, whereby the sign is always 0.

As a rule, BCD numbers have no sign. In conjunction with the conversion functions, the sign of a BCD number is stored in the leftmost (highest) decade, so that there is one less decade for the number.

When a BCD number is in a 16-bit word, the sign is in the uppermost decade, whereby only bit position 15 is relevant. Signal state “0” means that the number is positive. Signal state “1” stands for a negative number. The sign has no affect on the contents of the individual decades. An equivalent assignment applies for a 32-bit word.

The available value range is 0 to ± 999 for a 16-bit BCD number and 0 to ± 9 999 999 for a 32-bit number.

CHAR

A variable with data type CHAR (character) reserves one byte. Data type CHAR represents

Table 3.7 Special Characters for CHAR

CHAR	Hex	Description
\$\$	24 _{hex}	Dollar sign
\$'	27 _{hex}	Apostrophe
\$L or \$l	0A _{hex}	Line feed (LF)
\$P or \$p	0C _{hex}	New page (FF)
\$R or \$r	0D _{hex}	Carriage return (CR)
\$T or \$t	09 _{hex}	Tabulator

a single character in ASCII format. Example: “A”.

You can use any printable character in apostrophes. Some special characters require use of the notation shown in Table 3.7. Example: '\$\$' represents a dollar sign in ASCII code.

The MOVE function allows you to use two or four ASCII characters enclosed in apostrophes as a special form of data type CHAR for writing ASCII characters in a variable.

INT

A variable with data type INT is stored as an integer (16-bit fixed-point number). Data type INT has no special identifier.

A variable with data type INT reserves one word. The signal states of bits 0 to 14 represent the digit positions of the number; the signal state of bit 15 represents the sign (S). Signal state “0” means that the number is positive, signal state “1” that it is negative. A negative number is represented as two's complement. The permissible number range is

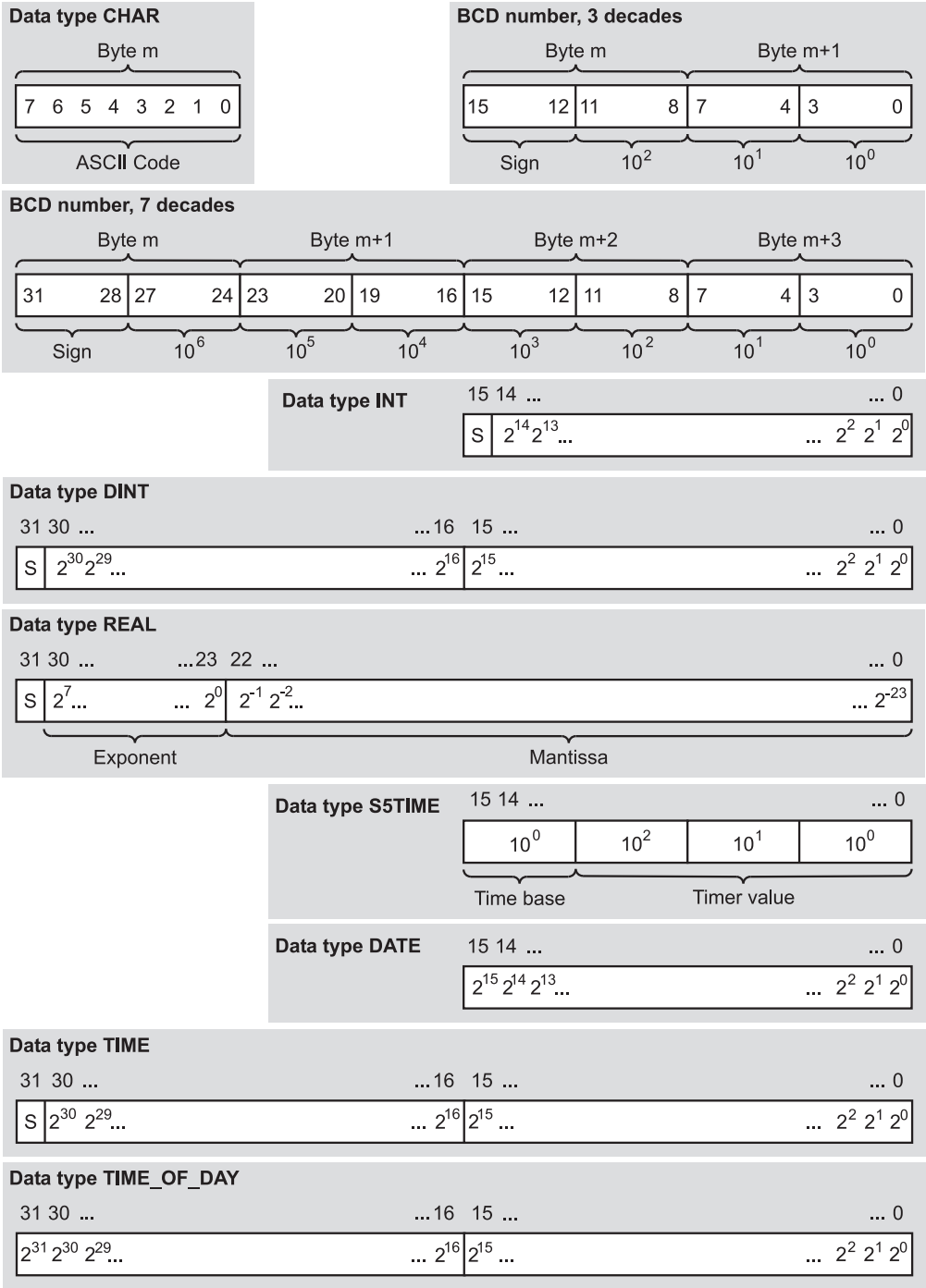
from +32 767 (7FFF_{hex})

to –32 768 (8000_{hex}).

DINT

A variable with data type DINT is stored as an integer (32-bit fixed-point number). An integer is stored as a DINT variable when it exceeds 32 767 or falls below –32 768 or when the number is preceded by type identifier L#.

A variable with data type DINT reserves one doubleword. The signal states of bits 0 to 30 represent the digit positions of the number; the sign is stored in bit 31. Bit 31 is “0” for a posi-



S Sign

Figure 3.15 Structure of the Variables of Elementary Data Type

tive and “1” for a negative number. Negative numbers are stored as two's complement. The number range is

from +2 147 483 647 (7FFF FFFF_{hex})

to -2 147 483 648 (8000 0000_{hex}).

REAL

A variable of data type REAL represents a fraction, and is stored as a 32-bit floating-point number. An integer is stored as a REAL variable when you add a decimal point and a zero.

In exponent representation, you can precede the “e” or “E” with an integer number or fraction with seven relevant digits and a sign. The digits that follow the “e” or “E” represent the exponent to base 10. STEP 7 handles the conversion of the REAL variable into the internal representation of a floating-point number.

REAL variables are divided into numbers, which can be represented with complete accuracy (“normalized” floating-point numbers) and those with limited accuracy (“denormalized” floating-point numbers).

The value range of a normalized floating-point number lies between:

$-3.402\,823 \times 10^{+38}$ to $-1.175\,494 \times 10^{-38}$
±0

$+1.175\,494 \times 10^{-38}$ to $+3.402\,823 \times 10^{+38}$

A denormalized floating-point number may be in the following range:

$-1.175\,494 \times 10^{-38}$ to $-1.401\,298 \times 10^{-45}$
and

$+1.401\,298 \times 10^{-45}$ to $+1.175\,494 \times 10^{-38}$

The S7-300 CPUs cannot calculate with denormalized floating-point numbers. The bit pattern of a denormalized number is interpreted as a zero. If a result falls within this range, it is represented as zero, and status bits OV and OS are set (overflow).

A variable of data type REAL consists internally of three components, namely the sign, the 8-bit exponent to base 2 and the 23-bit mantissa. The sign may assume the value “0” (positive) or “1” (negative). Before the exponent is stored, a constant value (bias, +127) is added to it so that it shows a value range of from 0 to 255. The mantissa represents the fractional por-

tion of the number. The integer portion of the mantissa is not saved, as it is either always 1 (in the case of normalized floating-point numbers) or always 0 (in the case of denormalized floating-point numbers). Table 3.8 shows the internal range of a floating-point number.

S5TIME

A variable with data type S5TIME is used in the basic languages STL, LAD and FBD to set the SIMATIC timers. It reserves one 16-bit word with 1 + 3 decades.

The time is specified in hours, minutes, seconds and milliseconds. STEP 7 handles conversion into internal representation. Internal representation is as BCD number in the range 000 to 999. The time interval can assume the following values: 10 ms (0000), 100 ms (0001), 1 s (0010), and 10 s (0011). The duration is the product of time interval and time value.

Examples:

S5TIME#500ms (= 0050_{hex})

S5T#2h46m30s (= 3999_{hex})

DATE

A variable with data type DATE is stored in a word as an unsigned fixed-point number. The contents of the variable correspond to the number of days since 01.01.1990. Its representation shows the year, month and day, separated from one another by a hyphen. Examples:

DATE#1990-01-01 (= 0000_{hex})

D#2168-12-31 (= FF62_{hex})

TIME

A variable with data type TIME reserves one doubleword. Its representation contains the information for days (d), hours (h), minutes (m), seconds (s) and milliseconds (ms), whereby individual items of this information may be omitted. The contents of the variable are interpreted in milliseconds (ms) and stored as a signed 32-bit fixed-point number. Examples:

TIME#24d20h31m23s647ms
(= 7FFF_FFFF_{hex})

TIME#0ms (= 0000_0000_{hex})

T#-24d20h31m23s648ms
(= 8000_0000_{hex})

Table 3.8 Range Limits of a Floating-Point Number

Sign	Exponent	Mantissa	Description
0	255	Not equal to 0	Not a valid floating-point number
0	255	0	+ infinite
0	1 ... 254	Arbitrary	Positive normalized floating-point number
0	0	Not equal to 0	Positive denormalized floating-point number
0	0	0	+ zero
1	0	0	– zero
1	0	Not equal to 0	Negative denormalized floating-point number
1	1 ... 254	Arbitrary	Negative normalized floating-point number
1	255	0	– infinite
1	255	Not equal to 0	Not a valid floating-point number

A “decimal representation” is also possible for TIME, e.g. TIME#2.25h or T#2.25h. Examples:

TIME#0.0h (= 0000_0000_{hex})
 TIME#24.855134d (= 7FFF_FFFF_{hex})

TIME_OF_DAY

A variable of data type TIME_OF_DAY reserves one doubleword. It contains the number of milliseconds since the day began (0:00 o'clock) in the form of an unsigned fixed-point number. Its representation contains the information for hours, minutes and seconds, separated by a colon. The milliseconds, which follow the seconds and are separated from them by a decimal point, may be omitted.

Examples:

TIME_OF_DAY#00:00:00 (= 0000_0000_{hex})
 TOD#23:59:59.999 (= 0526_5BFF_{hex})

3.5.5 Complex Data Types

STEP 7 defines the following four complex data types:

- ▷ DATE_AND_TIME
Date and time (BCD-coded)
- ▷ STRING
Character string with up to 254 characters
- ▷ ARRAY
Array variable (combination of variables of the same type)
- ▷ STRUCT
Structure variable (combination of variables of different types)

The data types are pre-defined, with the length of the data type STRING (character string) and the combination and size of the data types ARRAY and STRUCT (structure) being defined by the user.

Table 3.9 Examples of the Declaration of DT Variables and STRING Variables

Name	Type:	Initial Value	Comments
Date1	DT	DT#1990-01-01-00:00:00	DT variable minimum value
Date2	DATE_AND_TIME	DATE_AND_TIME#2089-12-31-23:59:59.999	DT variable maximum value
First_name	STRING[10]	'Jack'	STRING variable, 4 out of 10 char. specified
Last_name	STRING[14]	'Daniels'	STRING variable, all 7 char. specified
NewLine	STRING[2]	'\$R\$L'	STRING variable, special char. specified
BlankString	STRING[16]	"	STRING variable, no specification

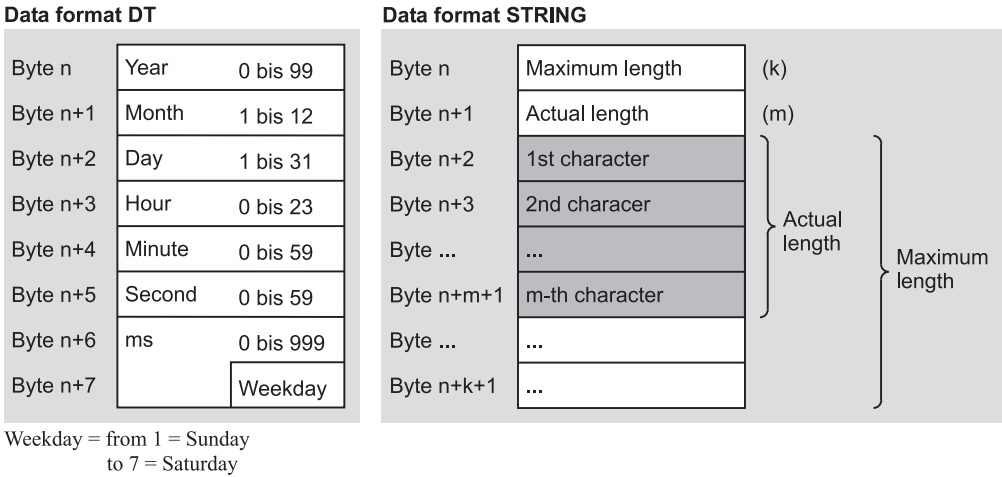


Figure 3.16 Structure of a DT and a STRING Variable

You can declare variables of complex data types only in global data blocks, in instance data blocks, as temporary local data or as block parameters.

Variables of complex data types can only be applied at block parameters as complete variables.

There are IEC functions for processing variables of data types DT and STRING, e.g. extraction of the date and conversion to the DATE representation or combining two character strings to one variable. These IEC functions are loadable standard FC blocks that you can find in the *Standard Library* under the *IEC Function Blocks* program.

DATE_AND_TIME

The data type DATE_AND_TIME represents a time consisting of the date and the time of day. You can also use the abbreviation DT in place of DATE_AND_TIME.

The individual components of a DT variable are ASCII coded (Figure 3.16).

STRING

The data type STRING represents a character string consisting of up to 254 characters. You specify the maximum permissible number of characters in square brackets following the keyword STRING.

This specification can also be omitted; the Editor then uses a length of 254 bytes. In the case of functions FCs, the Editor does not permit specification of the length or it demands the standard length of 254. A variable of data type STRING occupies two bytes more of memory than the declared maximum length.

Pre-assignment is carried out with ASCII-coded characters between single inverted commas or with a prefixed dollar sign in the case of certain characters (see data type CHAR).

If the initial or pre-assigned value is shorter than the declared maximum length, the remaining character locations are not reserved. When a variable of data type STRING is post-processed, only the currently reserved character locations are taken into consideration. It is also possible to define an “empty string” as the initial value. Figure 3.16 shows the structure of a STRING variable.

ARRAY

Data type ARRAY represents an array or field comprising a fixed number of elements of the same data type.

You specify the range of field indices in square brackets following the data type ARRAY. The initial value on the left must be less than or equal to the final value on the right. Both indices are INT numbers in the range –32,768 to

Table 3.10 Examples of an Array Declaration

Name	Type	Initial Value	Comments
MeasVal	ARRAY[1..24]	0.4, 1.5, 11 (2.6, 3.0)	Array variable with 24 REAL elements
	REAL		
TOD	ARRAY[-10..10]	21 (TOD#08:30:00)	TOD array with 21 elements
	TIME_OF_DAY		
Result	ARRAY[1..24,1..4]	96 (L#0)	Two-dimensional array with 96 elements
	DINT		
Character	ARRAY[1..2,3..4]	2 ("a"), 2 ("b")	Two-dimensional array with 4 elements
	CHAR		

+32,767. A field can have up to 6 dimensions each of whose limits are separated by a comma.

The data type of the individual field components is located in the line under the data type ARRAY. All data types except ARRAY are permissible; it can also be a user data type.

Pre-assignment

At the declaration stage, you can pre-assign values to individual field components (not as a block parameter in a function, as an in/out parameter in a function block or as a temporary variable). The data type of the pre-assignment value must match the data type of the field.

You do not require to pre-assign all field components; if the number of pre-assignment values is less than the number of field components, only the first components are pre-assigned. The number of pre-assignment values must not be greater than the number of field components. The pre-assignment values are each separated by a comma. Multiple pre-assignment with the same values is specified within round brackets with a preceding repetition factor.

Application

You can apply a field as a complete variable at block parameters of data type ARRAY with the same structure or at a block parameter of data type ANY. For example, you can copy the contents of a field variable using the system function SFC 20 BLKMOV. You can also specify individual field components at a block parameter if the block parameter is of the same data type as the components.

If the individual field components are of elementary data types, you can process them with "normal" LAD or FBD functions.

A field component is accessed with the field name and an index in square brackets. The index is a fixed value in LAD and FBD and cannot be modified at runtime (no variable indexing possible).

Multi-dimensional fields or arrays

Fields can have up to 6 dimensions. Multi-dimensional fields are analogous to one-dimensional fields. At the declaration stage, the ranges of the dimensions are written in square brackets, each separated by a comma.

Structure of the variables

An ARRAY variable always begins at a word boundary, that is, at a byte with an even address. ARRAY variables occupy the memory up to the next word boundary.

Components of data type BOOL begin in the least significant bit; components of data type BYTE and CHAR begin in the right-hand byte. The individual components are listed in order.

In multi-dimensional fields, the components are stored line-wise (dimension-wise) starting with the first dimension. With bit and byte components, a new dimension always starts in the next byte, and with components of other data types a new dimension always starts in the next word (in the next even byte).

STRUCT

The data type STRUCT represents a data structure consisting of a fixed number of components that can each be of a different data type.

You specify the individual structure components and their data types under the line with the variable name and the keyword STRUCT. All data types can be used including other structures.

Pre-assignment

At the declaration stage, you can pre-assign values to the individual structure components (not as a block parameter in a function, as an in/out parameter in a function block or as a temporary variable). The data types of the pre-assignment values must match the data types of the components.

Application

You can apply a complete variable at block parameters of data type STRUCT with the same structure or at a block parameter of data type ANY. For example, you can copy the contents of a STRUCT variable with the system function SFC 20 BLKMOV. You can also specify an individual structure component at a block parameter if the block parameter is of the same data type as the component.

If the individual structure components are of elementary data types, you can process them with “normal” LAD or FBD functions.

A structure component is accessed with the structure name and the component name separated by a dot.

Structure of the variables

A STRUCT variable always begins at a word boundary, that is, at a byte with an even address; following this, the individual components are located in the memory in the order of their declaration. STRUCT variables occupy the memory up to the next word boundary.

Components of data type BOOL begin in the least significant bit; components of data type BYTE and CHAR begin in the right-hand byte. Components of other data types begin at a word boundary.

A nested structure is a structure as a component of another structure. A nesting depth of up to 6 structures is possible. All components can be accessed individually with “normal” LAD or FBD functions provided they are of elementary data type. The individual names are each separated by a dot.

3.5.6 Parameter Types

Parameter types are data types for block parameters (Table 3.12). The length specifications in the Table refer to the memory requirements for block parameters for function blocks. You can also use TIMER and COUNTER in the symbol table as data types for timers and counters.

3.5.7 User Data Types

A user data type (UDT) corresponds to a structure (combination of components of any data type) with global validity. You can use a user data type if a data structure occurs frequently in your program or you want to assign a name to a data structure.

Table 3.11 Example of Declaring a Structure

Name	Type	Initial Value	Comment
MotCont	STRUCT		Simple structure variable with 4 components
On	BOOL	FALSE	Variable MotCont.On of type BOOL
Off	BOOL	TRUE	Variable MotCont.Off of type BOOL
Delay	S5TIME	S5TIME#5s	Variable MotCont.Delay of type S5TIME
maxSpeed	INT	5000	Variable MotCont.maxSpeed of type INT
	END_STRUCT		

Table 3.12 Overview of Parameter Types

Parameter Type	Description		Examples of Actual Addresses
TIMER	Timer	16 bits	T 15 or symbol
COUNTER	Counter	16 bits	Z 16 or symbol
BLOCK_FC	Function	16 bits	FC 17 or symbol
BLOCK_FB	Function block	16 bits	FB 18 or symbol
BLOCK_DB	Data block	16 bits	DB 19 or symbol
BLOCK_SDB	System data block	16 bits	(not used yet)
POINTER	DB pointer	48 bits	As pointer: P#M10.0 or P#DB20.DBX22.2 As address: MW 20 or I 1.0 or symbol
ANY	ANY pointer	80 bits	As range: P#DB10.DBX0.0 WORD 20 or any (complete) variable

UDTs have global validity; i.e., once declared, they can be used in all blocks. UDTs can be addressed symbolically; you assign the absolute address in the symbol table. The data type of a UDT (in the symbol table) is identical with the absolute address.

If you want to give a variable the data structure defined in the UDT, assign the UDT to it at declaration like a “normal” data type. The UDT can be absolutely addressed (UDT 0 to UDT 65,535) or symbolically addressed.

You can also define a UDT for an entire data type. When programming the data block, you assign this UDT to the block as a data structure.

The example “Message Frame Data” in Chapter 24.3 “Brief Description of the “Message Frame Example”” shows you how to work with user data types.

Creating UDTs

You can create a user data type using the SIMATIC Manager or also the program editor:

- ▷ In the SIMATIC Manager: select the *Blocks* object in the left part of the project window, and create a new UDT using INSERT → S7 BLOCK → DATA TYPE. You are then provided with the attributes window of the data

type. On the “General – Part 1” tab, enter the number (the absolute address UDTn) under “Name”. You can also enter the other block attributes later.

- ▷ In the program editor: use FILE → NEW to obtain a dialog box in which you can enter the desired data type (the absolute address UDTn) under “Object name”.

You can fill in the block header immediately when creating the data type, or enter the attributes later. With the data type open, you can program subsequent extensions in the program editor using FILE → PROPERTIES.

A double-click on data type *UDTn* in the SIMATIC Manager opens a declaration table that looks exactly like the declaration table of a data block. A UDT is programmed in exactly the same way as a data block, with individual lines for Name, Type, Initial value and Comments. The only difference is that switching to the data view is not possible. (With a UDT, you do not create any variables but only a collection of data types; for this reason, there can be no actual values here).

The initial values you program in the UDT are transferred to the variables at declaration.

Basic Functions

This section of the book describes those functions of the LAD and FBD programming languages which represent a certain “basic functionality”. These functions allow you to program a PLC on the basis of contactor or relay controls.

In a ladder diagram (LAD), the arrangement of the contacts in **series and parallel circuits** determines the combining of binary signal states. In a function block diagram (FBD), boxes analogous to electronic switching systems represent the **boolean functions** AND and OR.

The **memory functions** hold onto an RLO so that it can, for example, be scanned and processed further in another part of the program.

The **move functions** are used to exchange the values of individual operands and variables or to copy entire data areas.

The timing relays in contactor control systems are **timers** in programmable controllers. The timers integrated in the CPU allow you, for example, to program waiting and monitoring times.

Finally, the **counters** can count up and down in the range 0 to 999.

This section of the book describes the functions for the operand areas for inputs, outputs, and memory bits. Inputs and outputs are the link to the process or plant. The memory bits correspond to auxiliary contactors which store binary states. The subsequent sections of the book describe the remaining operand areas, on which you can also use binary logic. Essentially, these are the data bits in global data blocks as well as the temporary and static local data bits.

In Chapter 5 “Memory Functions”, you will find a programming example for the binary logic operations and memory functions, and in Chapter 8 “Counters”, an example for timers and counters. In both cases, the example is in an FC function without block parameters. You will find the same examples as function blocks (FBs) with block parameters in Chapter 19 “Block Parameters”.

4 Binary Logic Operations

Series and parallel circuits (LAD), AND, OR and exclusive OR functions (FBD): negation; taking account of the sensor type

5 Memory functions

LAD coils; FBD boxes; midline outputs; edge evaluation; conveyor belt example

6 Move functions

MOVE box, system functions for moving data

7 Timers

Starting 5 different kinds of timer, resetting and scanning a timer; IEC timers

8 Counters

Setting a counter; up and down counting; resetting and scanning a counter; IEC counters; feed example

4 Binary Logic Operations

4.1 Series and Parallel Circuits (LAD)

Binary signal states are combined in LAD through series and parallel connection of contacts. Series connection corresponds to an AND function and parallel connection to an OR function. You use the contacts to check the signal states of the following binary operands:

- ▷ Input and output bits, memory bits
- ▷ Timers and counters
- ▷ Global data bits
- ▷ Temporary local data bits
- ▷ Static local data bits
- ▷ Status bits (evaluation of calculation results)

You can reference an operand via a contact using either an absolute or a symbolic address. LAD uses only NO contacts (scan for signal state “1”) and NC contacts (scan for signal state “0”).

A rung may consist of a single contact, but it may also consist of a large number of contacts connected together. A rung must always be terminated, for example with a coil. The coil controls a binary operand with the RLO (the “power flow”) of the rung.

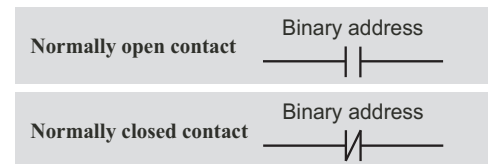
The examples shown in this chapter can be found in function block FB 104 of the “Basic Functions” program in the “LAD_Book” library that you can download from the publisher's Website (see page 8).

For incremental programming, you will find the elements for binary logic operations in the Program Element Catalog (with VIEW → OVERVIEWS [Ctrl -K] or with INSERT → PROGRAM ELEMENTS) under “Bit Logic”.

4.1.1 NO Contact and NC Contact

In order to explain the bit logic combinations in a ladder diagram, we will refer below as graphically as possible to “contact closed”, “power flowing”, and “coil energized”. If “power” is flowing at a point in the ladder diagram, this means that the bit logic combination applies up to this point; the result of the logic operation (RLO) is “1”. If “power” is flowing in a single coil, the coil is energized; the associated binary operand then carries signal state “1”.

LAD has two kinds of contacts for scanning bit operands: the NO contact and the NC contact.



Normally open (NO) contact

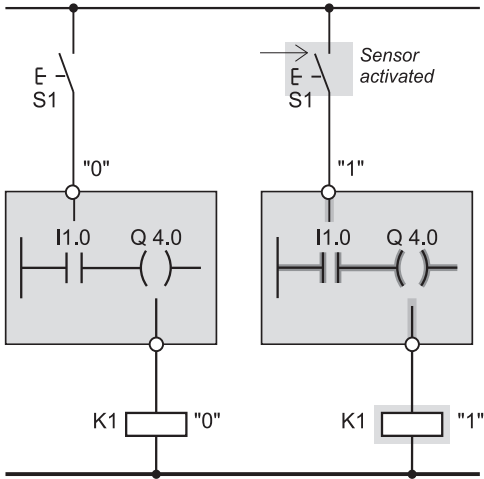
A normally open contact corresponds to a scan for signal state “1”. If the scanned binary operand has signal state “1”, the NO contact is activated, so it closes and “power flows”.

The example in Figure 4.1 (left side) shows sensor S1 connected to input I 1.0 and scanned by an NO contact. If sensor S1 is open, input I 1.0 is “0” and no power flows through the NO contact. Contactor K1, controlled by output Q 4.0, does not switch on.

If sensor S1 is now activated, input I 1.0 has signal state “1”. Power flows from the left power rail through the NO contact into the coil, and contactor K1, which is connected to output Q 4.0, is activated.

The NO contact scans the input for signal state “1” and then closes, regardless of whether the sensor at the input is an NO or NC contact.

Mode of operation of NO contact



Mode of operation of NC contact

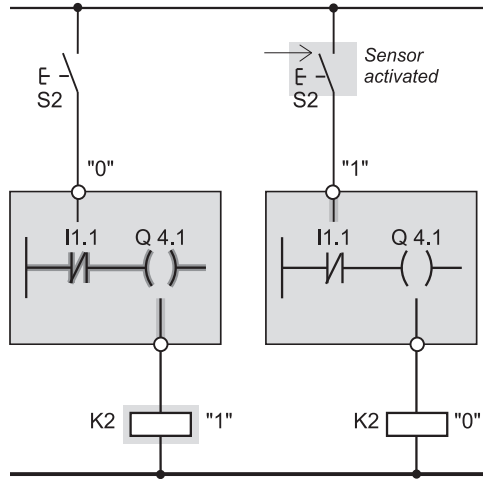


Figure 4.1 NO Contacts and NC Contacts

Normally closed (NC) contact

Power flows through an NC contact if the binary operand has the signal state “0”. If the signal state is “1”, an NC contact “opens” and the flow of power is interrupted.

In the example in Figure 4.1 (right side), power flows through the NC contact if sensor S2 is not closed (input I 1.1 has signal state “0”). Power also flows in the coil and energizes contactor K2 at output Q 4.1.

If sensor S2 is now activated, input I 1.1 has signal state “1” and the NC contact opens. The power flow is interrupted and contactor K2 releases.

The NC contact checks the input for signal state “0” and then remains closed, regardless of whether the sensor at the input is an NO or NC contact (also see Chapter 4.3 “Taking Account of the Sensor Type”).

4.1.2 Series Circuits

In series circuits, two or more contacts are connected in series. Power flows through a series circuit when all contacts are closed.

Figure 4.2 shows a typical series circuit. In network 1, the series circuit has three contacts; any binary operands can be scanned. All contacts are NO contacts. If the associated operands all have signal state “1” (that is, if the NO contacts

are activated), power flows through the rung to the coil. The operand controlled by the coil is set to “1”. In all other cases, no power flows and the operand *Coil1* is reset to “0”.

Network 2 shows a series circuit with one NC contact. Power flows through an NC contact if the associated operand has signal state “0” (that is, the NC contact is not activated). So power only flows through the series circuit in the example if the operand *Contact4* has signal state “1” and the operand *Contact5* has signal state “0”.

4.1.3 Parallel Circuits

When two or more contacts are arranged one under the other, we refer to a parallel circuit. Power flows through a parallel circuit if one of the contacts is closed.

Figure 4.2 shows a typical parallel circuit. In network 3, the parallel circuit consists of three contacts; any binary operands can be scanned. All contacts are NO contacts. If one of the operands has signal state “1”, power flows through the rung to the coil. The operand controlled by the coil is set to “1”. If all operands scanned have signal state “0”, no power flows to the coil and the operand *Coil3* is reset to “0”.

Network 4 shows a parallel circuit with one NC contact. Power flows through an NC contact if the associated operand is “0”, that is, power

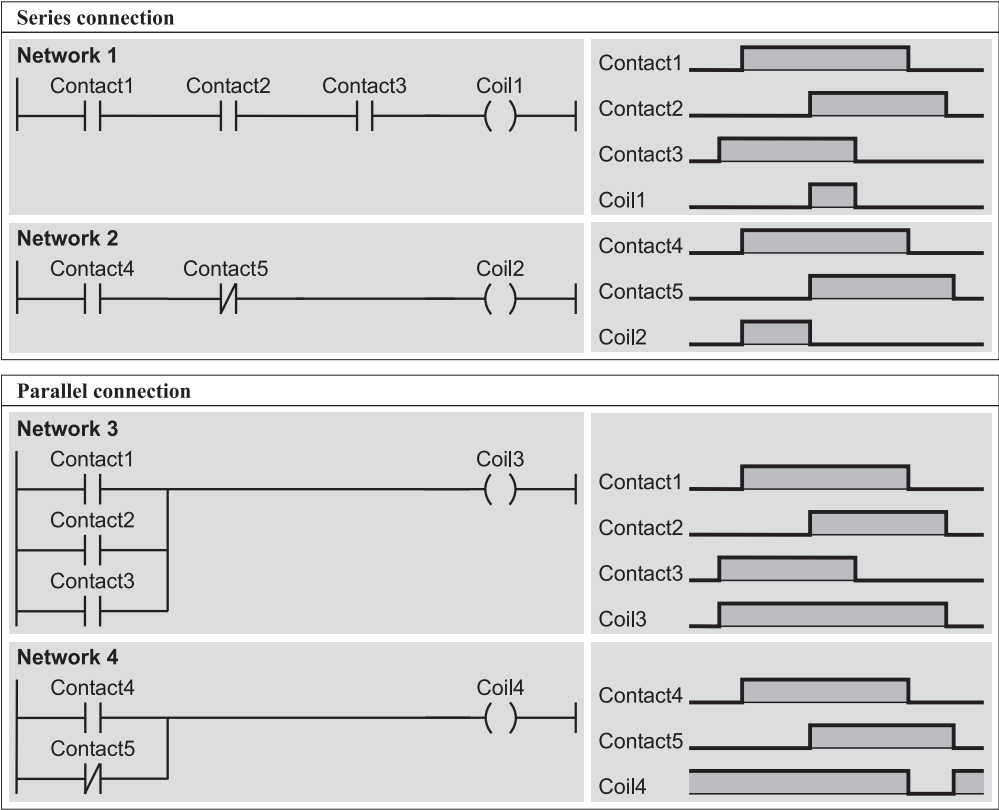


Figure 4.2 Series and Parallel Circuits

flows through the series circuit in the example if the operand *Contact4* has signal state “1” or the operand *Contact5* has signal state “0”.

LAD, you can also program a branch in the middle of the rung (for an example, see Figure 4.3 network 8). You then get a parallel branch that does not begin at the left power rail. Use of LAD program elements is restricted to this parallel branch; your attention is drawn to this in the relevant chapters.

An “open” parallel circuit is called a “T-branch”.

4.1.4 Combinations of Binary Logic Operations

You can combine series and parallel circuits, for example, by arranging several series circuits in parallel or several parallel circuits in series.

You can combine series and parallel circuits even when both types are complex in nature (Figure 4.3).

Connecting series circuits in parallel

Instead of contacts, you can also arrange series circuits one under the other. Figure 4.3 shows two examples. In network 5, power flows into the coil if *Contact1* and *Contact2* are closed or if *Contact3* and *Contact4* are closed. In the lower rung (network 6), power flows if *Contact5* or *Contact6* and *Contact7* or *Contact0* are closed.

Connecting parallel circuits in series

Instead of contacts, you can also arrange parallel circuits in series. Figure 4.3 shows two examples. In network 7, power flows into the coil if either *Contact1* or *Contact3* and either

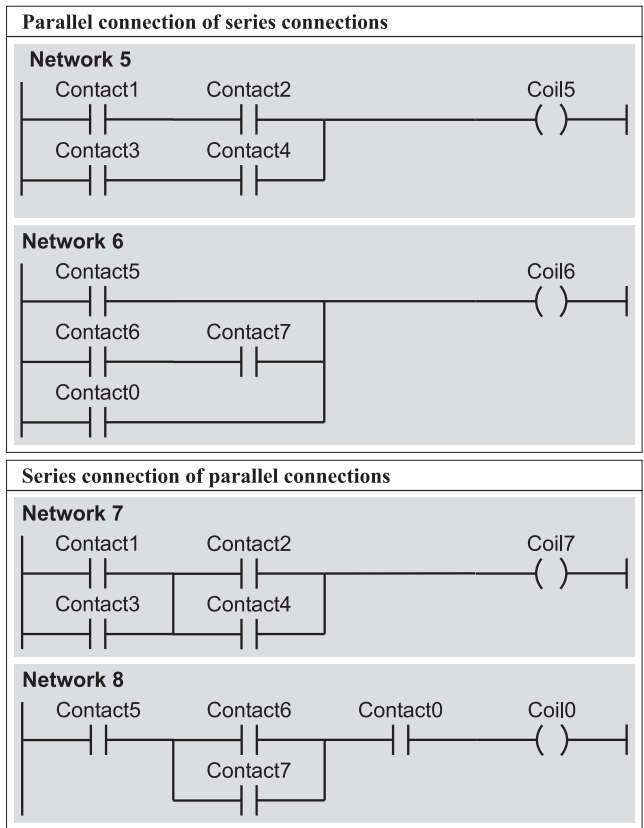


Figure 4.3 Series and Parallel Circuits in Combination

Contact2 or Contact4 are closed. To allow power to flow in the lower example (network 8), Contact5, Contact0 and either Contact6 or Contact7 must be closed.

4.1.5 Negating the Result of the Logic Operation

The NOT contact negates the RLO. You can use this contact, for example, to run a series circuit negated to a coil (Figure 4.4 Network 9). Power will then only flow into the coil if there is no power in the NOT contact, that is, if either Contact1 or Contact2 is open (see the Figure in the adjacent pulse diagram).

The same applies by analogy for network 10, in which a NOT contact is inserted after a parallel circuit. Here, Coil10 is set if neither of the contacts is closed.

You can insert NOT instead of another contact into a branch that begins at the left power rail. Positioning of the NOT contact is not permissible directly on the left power rail or in a parallel branch which commences in the middle of the rung.

4.2 Binary Logic Operations (FBD)

In FBD, the logic operations performed on binary signal states take the form of AND, OR and Exclusive OR functions. The operands whose signal states you want to scan and combine are written at the inputs of these functions. You can scan the following operands:

- ▷ Input and output bits, memory bits (discussed in this section)

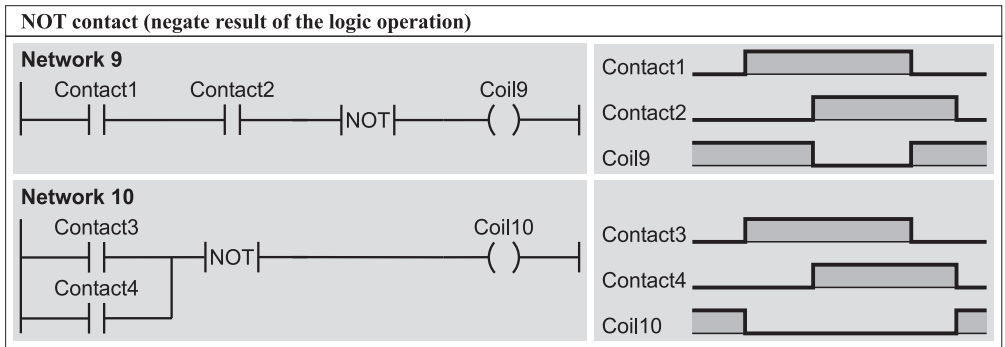


Figure 4.4 Examples of a NOT Contact

- ▷ Timers and counters
- ▷ Global data bits
- ▷ Temporary local data bits
- ▷ Static local data bits
- ▷ Status bits (evaluation of calculation results)

Every binary operand can be addressed absolutely or symbolically. When scanning a binary operand, or within a binary logic circuit, you can negate the result of the logic operation with the negation symbol (which is a circle).

In FBD, you program one binary logic circuit per network. The logic circuit may consist of only one or of a very large number of interconnected functions. A logic circuit, or logic operation, must always be terminated, for example with an assign statement. The assign controls a binary operand with the result of the logic operation.

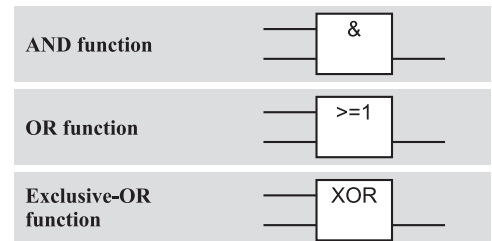
The examples shown in this chapter can be found in function block FB 104 of the “Basic Functions” program in the “FBD_Book” library that you can download from the publisher's Website (see page 8).

For incremental programming, you will find the elements for binary logic operations in the Program Element Catalog (VIEW → OVERVIEWS [Ctrl - K] or with INSERT → PROGRAM ELEMENTS) under “Bit Logic”.

4.2.1 Elementary Binary Logic Operations

FBD uses the binary functions AND, OR, and Exclusive OR. All functions may (theoretically) have any number of function inputs. If an

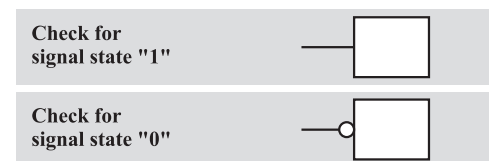
input leads directly to the function element, the signal state of the operand scanned is used directly in the logic operation; if the input has a negation character (a circle), the signal state of the scanned operand is negated prior to execution of the logic operation (see below).



The number of binary functions and the scope of a binary function are theoretically unlimited; in practice, however, limits are set by the length of a block or the size of the CPU's main memory.

Scanning and assigning signal states

Before the binary functions perform logic operations on signal states, they scan the binary operands at the function inputs. An operand can be scanned for “1” or “0”. If scanned for “1”, the function input leads directly to the box. A scan for “0” is recognizable by the negation character at the function input.



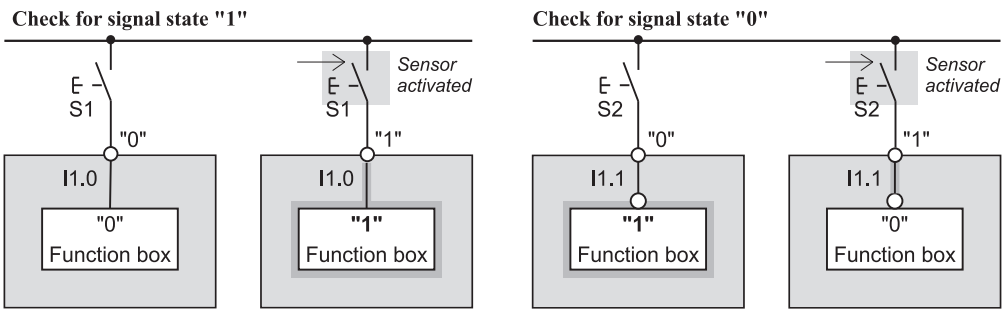


Figure 4.5 Scanning for Signal State "1" and "0"

A scan for "1" produces a scan result of "1" when the signal state of the binary operand scanned is "1"; it produces a scan result of "0" when the signal state of the binary operand is "0". A scan for signal state "0" negates the scan result, that is, the scan result is "1" when the status of the binary operand scanned is "0". The binary functions combine the *scan result*, which is, at it were, the result applied "directly" to the box. As far as functionality is concerned, these two methods of scanning binary operands allow you to treat NO contacts and NC contacts identically.

Here an example: "0" is applied to the input module for a non-activated NO contact (Figure 4.5). A scan for signal state "1" forwards this status to a function box. To effect the same for an NC contact, you have to scan an input with an NC contact for signal state "0" (must include a circle for negation). The signal state "1" applied to the input module for a non-activated NC contact is then converted into signal state "0" at the function box.

If you now activate both the NO and NC contacts, the function box will show signal state "1" in both cases. Additional information can be found in Chapter 4.3 "Taking Account of the Sensor Type").

You must always connect the output of a binary function; in the simplest case, simply connect the output to an Assign box (also see Chapter 5 "Memory Functions"). With this result of the logic operation, you can also start a timer, execute a digital operation, call a block, and so on. The next chapter provides all the information you need.

To assign the signal state of a binary operand directly to another binary operand without performing any additional logic operations, for example to connect an input directly to an output, the AND function is normally used, although it would also be possible to use an OR or Exclusive OR.



Simply select the AND function, connecting only one function input and removing the other.

AND function

The AND function combines two binary states with one another and produces an RLO of "1" when both states (both scan results) are "1". If the AND function has several inputs, the scan results of all inputs must be "1" for the collective RLO to be "1". In all other cases, the AND function produces an RLO of "0" at its function output.

Figure 4.6 shows an example of an AND function. In Network 1, the AND function has three inputs, each of which can be connected to any binary operand. All operands are scanned for signal state "1", so that the signal state of the operands is directly ANDed. If all the operands that were scanned have a signal state of "1", the AND function sets the operand *Output1* to "1" via the Assign box (see next chapter). In all other cases, the AND condition is not fulfilled and operand *Output1* is reset to "0".

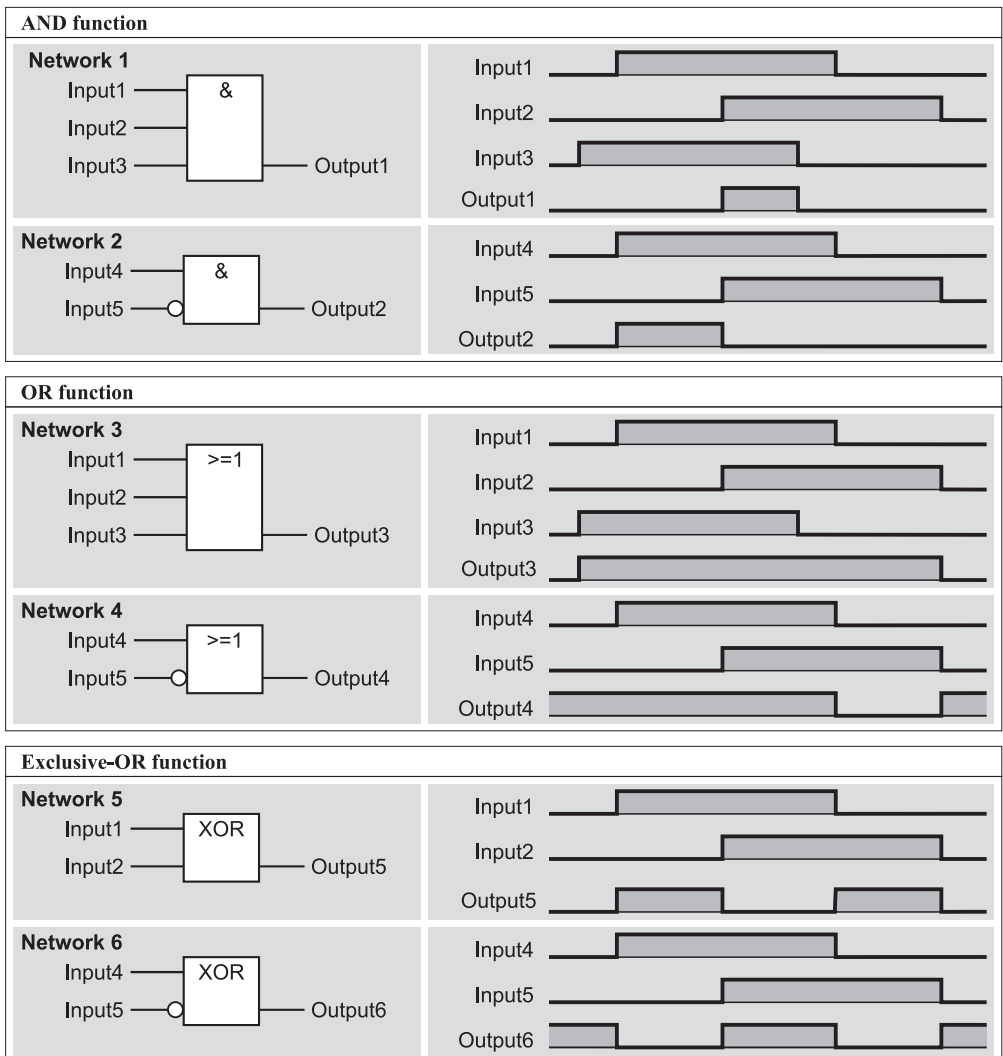


Figure 4.6 Examples of Binary Functions

Network 2 shows an AND function with a negated input. Negation of the input is indicated by a circle. The scan result for a negated operand is “1” when this operand is “0”, that is, the AND condition in the example is fulfilled when the operand *Input4* is “1” and the operand *Input5* is “0”.

OR function

The OR function combines two binary signal states and returns an RLO of “1” when one of

these states (one of the scan results) is “1”. If the OR function has several inputs, the scan result of only one input need be “1” in order for the result of the logic operation (RLO) to be “1”. The OR function returns an RLO of “0” when the scan results of all inputs are “0”.

Figure 4.6 shows an example of an OR function. In Network 3, the OR function has three inputs; each of these inputs may be connected to any binary operand. All operands are scanned for signal state “1”, so that the signal

state of the operands is directly ORed. If one or more of the operands scanned have signal state “1”, the next statement sets the operand *Output1* to “1”. If all of the operands scanned have signal state “0”, the OR condition is not fulfilled and operand *Output1* is reset to “0”.

Network 4 shows an OR function with a negated input. Negation is represented by a circle. The scan result of a negated operand is “1” when that operand is “0”, that is, the OR condition in the example is fulfilled when the operand *Input4* has a signal state of “1” or the operand *Input5* has a signal state of “0”.

Exclusive OR function

The Exclusive OR function combines two binary states with one another and returns an RLO of “1” when the two states (scan results) are not the same, and RLO “0” when the two states (scan results) are identical.

Figure 4.6 shows an example of an Exclusive OR function. In Network 5, two inputs, both of which are scanned for signal state “1”, lead to the Exclusive OR function. If only one of the operands scanned is “1”, the Exclusive OR condition is fulfilled and the operand *Output1* is set to “1”. If both operands are “1” or “0”, operand *Output1* is reset to “0”.

Network 6 shows an Exclusive OR function with a negated input. Negation is represented

by a circle. The scan result of a negated operand is “1” when that operand is “0”, that is, the Exclusive OR condition in the example is fulfilled when both input operands have the same signal state.

You can also program an Exclusive OR function with more than two inputs, in which case the Exclusive OR condition is fulfilled (in the case of a direct scan) when an uneven number of the operands scanned have a scan result of “1”.

4.2.2 Combinations of Binary Logic Operations

You can easily combine binary functions with one another. For instance, you can combine several AND functions into one OR function or two OR functions into one Exclusive OR function. The number of functions per logic operation (per network) is theoretically unlimited.

The use of a “T-branch” in a logic operation gives you additional options, allowing you to program more than one output per logic operation (see Chapter 5.2 “FBD Boxes”).

You can link the output of one binary function with the input of another binary function in order to implement complex binary logic operations. Figure 4.7 provides a number of examples.

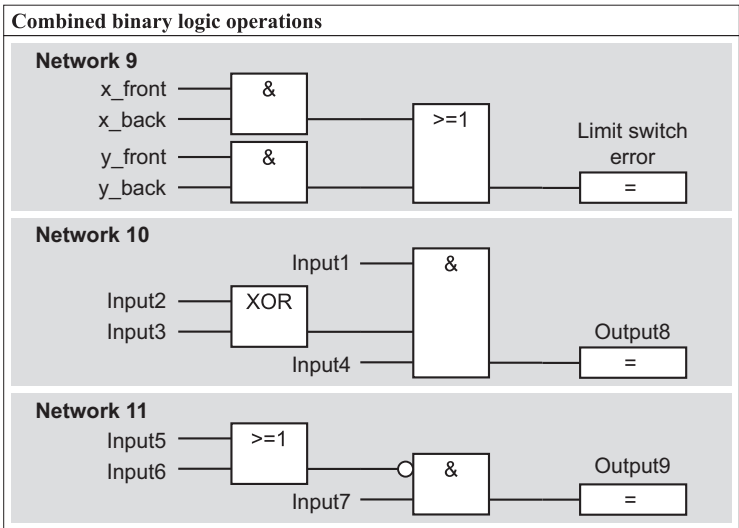


Figure 4.7 Examples of Binary Logic Operations in Combination

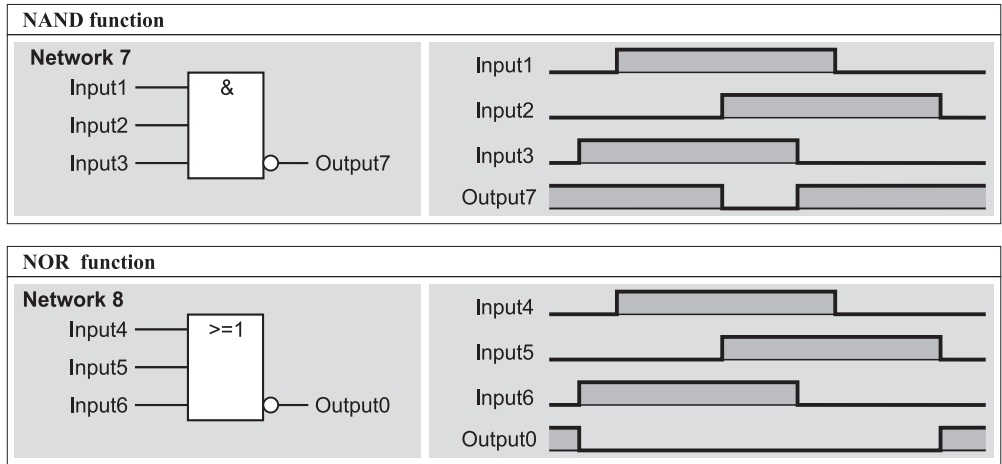


Figure 4.8 NAND and NOR Functions

Network 9: You are monitoring the limit switches at the ends of an X axis and a Y axis. These limit switches may not be actuated in pairs; otherwise, *limit switch error* will be reported.

Network 10: You can link arbitrary function inputs with binary functions, for example you can place an Exclusive OR function in front of the second input of an AND function

Network 11: Using the negation symbol, you negate the RLO, even between binary functions, for instance you can negate the RLO of an OR function and use it as input to an AND function.

4.2.3 Negating the Result of the Logic Operation

The circle at the input or output of a function symbol negates the result of the logic operation. You can use negation

- ▷ to scan a binary operand, which is equivalent to scanning for signal state “0” (see above),
- ▷ between two binary functions (which is equivalent to negating the result of the logic operation), or
- ▷ at the output of a binary function (for example if you want to set or reset a binary operand and when the condition is not fulfilled, that is, when RLO = “0”).

A negation may not immediately follow a T-branch.

Figure 4.8 shows a NAND function (an AND function with negated output) and a NOR function (an OR function with negated output). The RLO of a NAND function is “0” only when all inputs have a signal state of “1”. A NOR function returns an RLO of “1” only when none of the inputs has a signal state of “1”.

4.3 Taking Account of the Sensor Type

When scanning a sensor in a user program, you must take account of whether the sensor is an NO contact or an NC contact. Depending on the sensor type, there is a different signal state at the relevant input when the sensor is activated: “1” for an NO contact and “0” for an NC contact. The CPU has no means of determining whether an input is occupied by an NO contact or by an NC contact. It can only recognize the signal state “1” or “0”.

Programming with LAD

If you structure the program in such a way that you want a scan result of “1” when a sensor is activated in order to combine that scan result further, you must scan the input differently for different kinds of sensors. NO contacts and NC

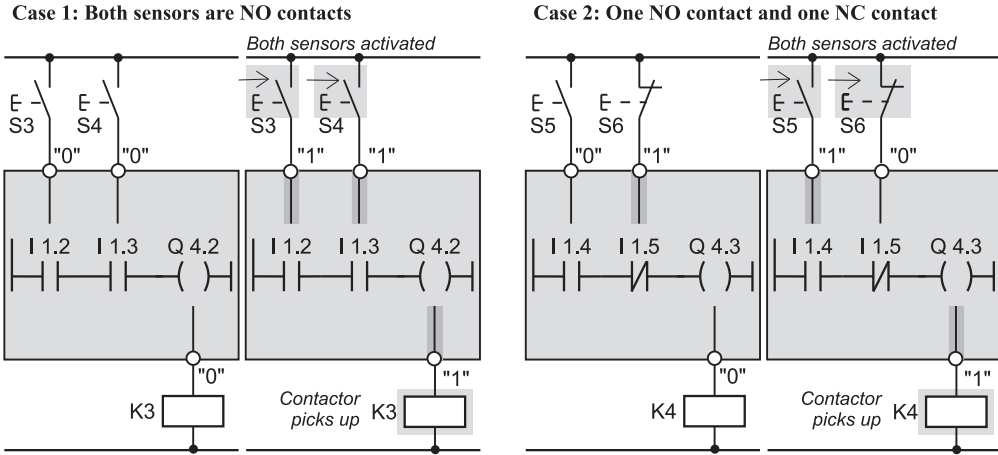


Figure 4.9 Taking Account of the Sensor Type (LAD)

contacts are available to you for this purpose. An NO contact return “1” if the scanned input is also “1”. An NC contact returns “1” if the scanned input is “0”. In this way, you can also directly scan inputs that are to trigger activities when they are “0” (zero-active inputs) and subsequently re-gate the scan result.

The example in Figure 4.9 shows programming dependent on the sensor type. In the first case, two NO contacts are connected to the programmable controller, and in the second case one NO contact and one NC contact. In both cases, a contactor connected to an output is to pick up if both sensors are activated. If an NO contact is activated, the signal state at the input is “1”, and this is scanned with an NO contact so that power can flow when the sensor is activated. If both NO contacts are activated, power flows through the rung to the coil and the contactor picks up.

If an NC contact is activated, the signal state at the input is “0”. In order to have power flow in this case when the sensor is activated, the result must be scanned with an NC contact. Therefore, in the second case, an NO contact and an NC contact must be connected in series to make the contactor pick up when both sensors are activated.

Programming with FBD

If you structure the program in such a way that you want a scan result of “1” when a sensor is

activated in order to combine that scan result further, you must scan the input differently for different kinds of sensors. An NO contact produces signal state “1” when activated, and is scanned directly when activation of the sensor is to produce a scan result of “1”. An NC contact returns signal state “0” when activated; if you want a scan result of “1” when the NC contact is activated, it must be negated, then scanned. In this way, you can also scan inputs that are to trigger activities even when they have a signal state of “0” (zero-active inputs) and further combine the scan result.

The example in Figure 4.10 shows sensor type-dependent programming. In the first case, two NO contacts are connected to the programmable controller, and in the second case one NO contact and one NC contact. In both cases, a contactor connected to an output is to pick up if both sensors are activated. If an NO contact is activated, the signal state at the input is “1”, and this is scanned directly so that the scan result is “1” when the sensor is activated. If both NO contacts are activated, and AND condition is fulfilled and the contactor picks up.

If an NC contact is activated, the signal state at the input is “0”. In order to obtain a scan result of “1”, the input must be negated when scanned. In the second case, you need an AND function with one direct and one negated input in order for the contactor to pick up when both sensors are activated.

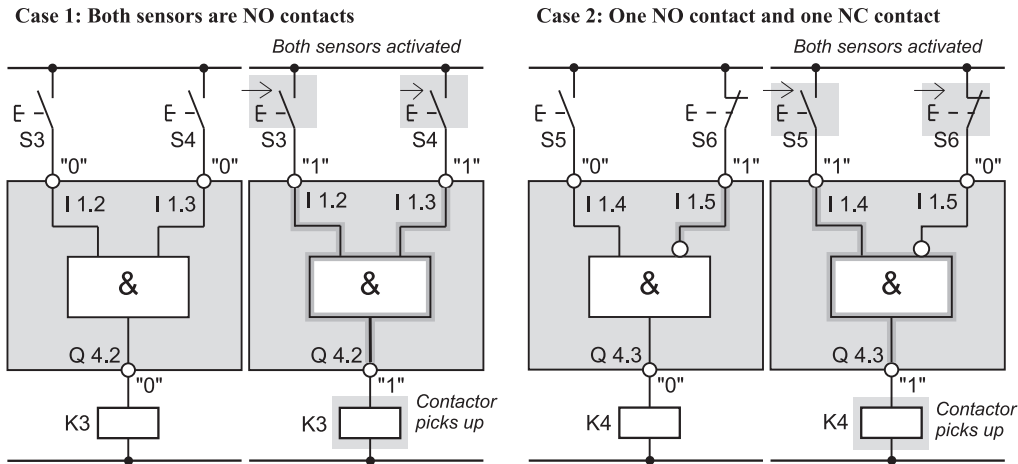


Figure 4.10 Taking Account of the Sensor Type (FBD)

5 Memory Functions

5.1 LAD Coils

In a ladder diagram (LAD), the memory functions are used in conjunction with series and parallel circuits in order to influence the signal states of the binary operands with the aid of the result of the logic operation (RLO) generated in the CPU.

The following memory functions are available

- ▷ The single coil as an assignment of the RLO
- ▷ The coils S and R as individually programmed memory functions
- ▷ The boxes RS and SR as memory functions
- ▷ The midline outputs as intermediate buffers
- ▷ The coils P and N as edge evaluations of the power flow
- ▷ The boxes POS and NEG as edge evaluations of operands

Midline outputs and edge evaluations are discussed in detail in subsequent chapters.

You can use the memory functions described in this chapter in conjunction with all binary operands. There are restrictions when using temporary local data bits as edge memory bits.

The examples shown in this chapter are found in function block 105 of the “Basic Functions” program in the “LAD_Book” library that you can download from the publisher's Website (see page 8).

For incremental programming, you will find the program elements for the memory functions in the program element catalog (with VIEW → OVERVIEWS [Ctrl - K] or with INSERT → PROGRAM ELEMENTS) under “Bit Logic”.

5.1.1 Single Coil

The single coil as terminator of a rung assigns the power flow directly to the operand located

at the coil. The function of the single coil depends on the Master Control Relay (MCR): If the MCR is activated, signal state “0” is assigned to the binary operand located over the coil.



If power flows into the coil, the operand is set; if there is no power, the operand is reset (Figure 5.1 Network 1). With a NOT contact before the coil, you reverse the function (Network 2).

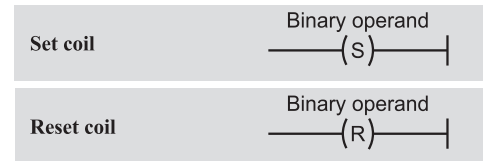
You can also direct the power flow into several coils simultaneously by arranging the coils in parallel with the help of a T-branch (Network 3). All operands specified over the coils respond in the same way. Up to 16 coils can be connected in parallel.

You can arrange further contacts in series and parallel circuits after the T-branch and before the coil (Network 4).

See Chapter 4.1 “Series and Parallel Circuits (LAD)”, for further examples of the single coil.

5.1.2 Set and Reset Coil

Set and reset coils also terminate a rung. These coils only become active when power flows through them.



If power flows in the set coil, the operand over the coil is set to signal state “1”. If power flows in the reset coil, the operand over the coil is reset to signal state “0”. If there is no power in the set or reset coil, the binary operand remains unaffected (Figure 5.1, Networks 5 and 6).

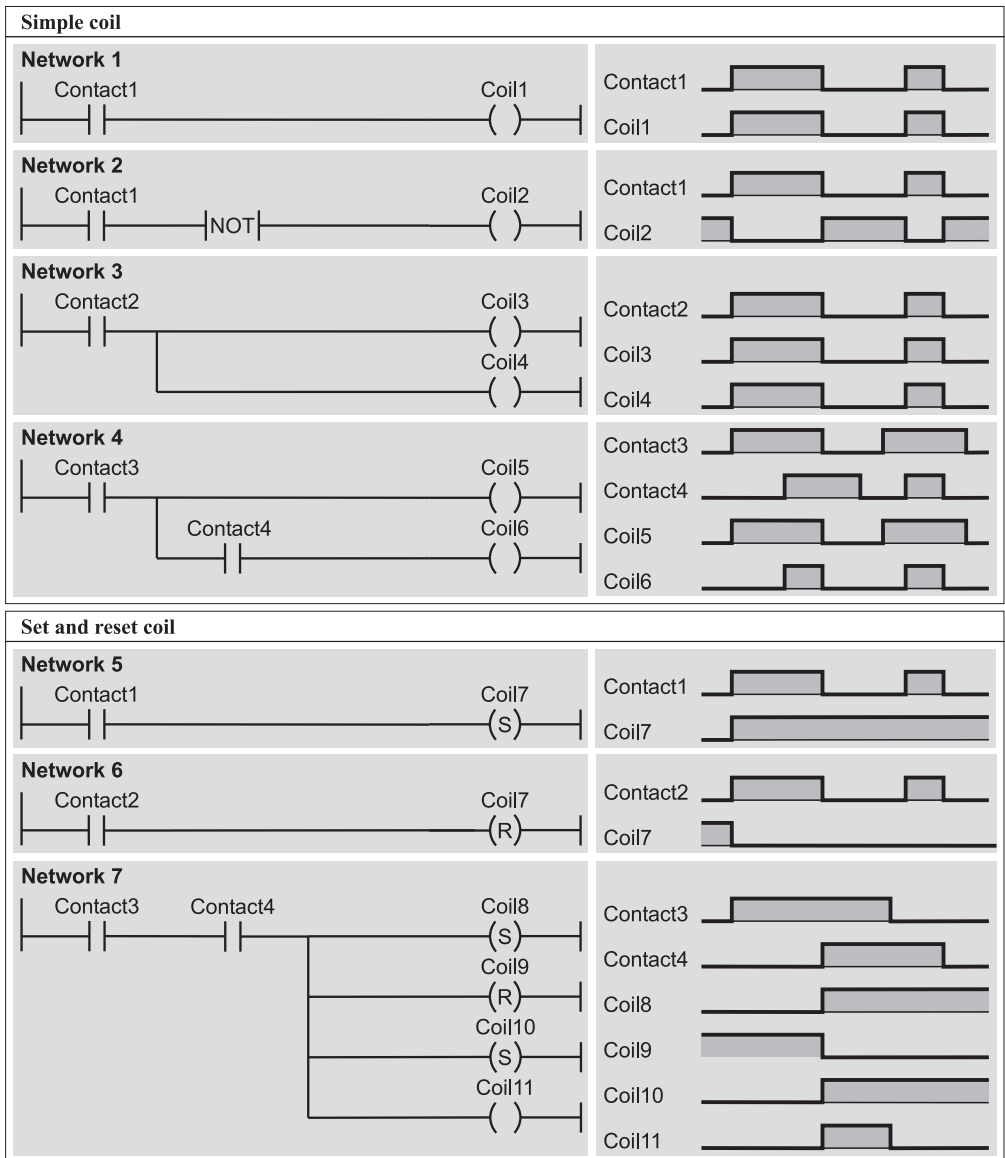


Figure 5.1 Single Coil, Set and Reset Coil

The function of the set and reset coils depends on the Master Control Relay. If the MCR is activated, the binary operand over the coil is not affected.

Please note that the operand used with a set or reset coil is usually reset at startup (complete restart). In special cases, the signal state is retained: This depends on the startup mode (for

example warm restart), on the operand used (for example static local data), and on the settings in the CPU (for example retentive characteristics).

You can arrange several set and reset coils in any combination and together with single coils in the same rung (Network 7). To achieve clarity in your programming, it is advisable to

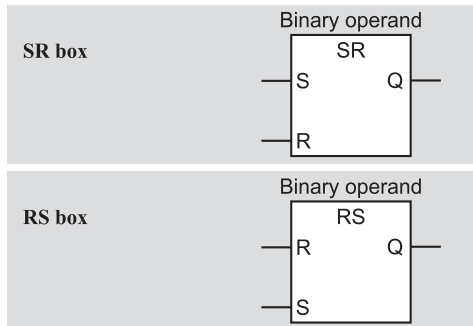
group the set and reset coils affecting an operand together in pairs, and to use them only once in each case. You should also avoid additionally controlling these operands with a single coil.

As with the single coil, you can also arrange contacts after the branch and before a set and reset coil.

5.1.3 Memory Box

The functions of the set and reset coil are summarized in the box of a memory function. The common binary operand is located over the box. Input S of the box corresponds here to the set coil, input R to the reset coil. The signal state of the binary operand assigned to the memory function is at output Q of the memory function.

There are two versions of the memory function: As SR box (reset priority) and RS box (set priority). Apart from the labeling, the boxes also differ from each other in the arrangement of the S and R inputs.



A memory function is set (or, more precisely, the binary operand over the memory box is set) if the set input has signal state “1” and the reset input has signal state “0”. A memory function is reset if there is a “1” at the reset input and a “0” at the set input. Signal state “0” at both inputs has no effect on the memory function. If both inputs are “1” at the same time, the two memory functions respond differently: the SR memory function is reset and the RS memory function is set.

The function of the memory box depends on the Master Control Relay. If the MCR is active, the binary operand of a memory box is no longer affected.

Please note that the operand used with a memory function at startup (complete restart) is usually reset. In special cases, the signal state of a memory box is retained. This depends on the startup mode (for example warm restart), on the operand used (for example static local data), and on the settings in the CPU (for instance retentive characteristics).

SR memory function

In the SR memory box, the reset input has priority. Reset priority means that the memory function is or remains reset if power flows “simultaneously” in the set input and the reset input. The reset input has priority over the set input (Figure 5.2, Network 8).

Because the statements are executed in sequence, the CPU initially sets the memory operand because the set input is processed first, but resets it again when it processes the reset input. The memory operand remains reset while the rest of the program is processed.

If the memory operand is an output, this brief setting only takes place in the process-image output table, and the (external) output on the relevant output module remains unaffected. The CPU does not transfer the process-image output table to the output modules until the end of the program cycle.

The memory function with reset priority is the “normal” form of the memory function, since the reset state (signal state “0”) is normally the safer or less hazardous state.

RS memory function

In the RS memory box, the set input has priority. Set priority means that the memory function is or remains set if power flows “simultaneously” in the set input and the reset input. The set input then has priority over the reset input (Figure 5.2 Network 9).

In accordance with the sequential execution of the instructions, the CPU resets the memory operand with the reset input first processed, but then sets it again when processing the set input. The memory operand remains set while the rest of the program is processed.

If the memory operand is an output, this brief resetting takes place only in the process-image

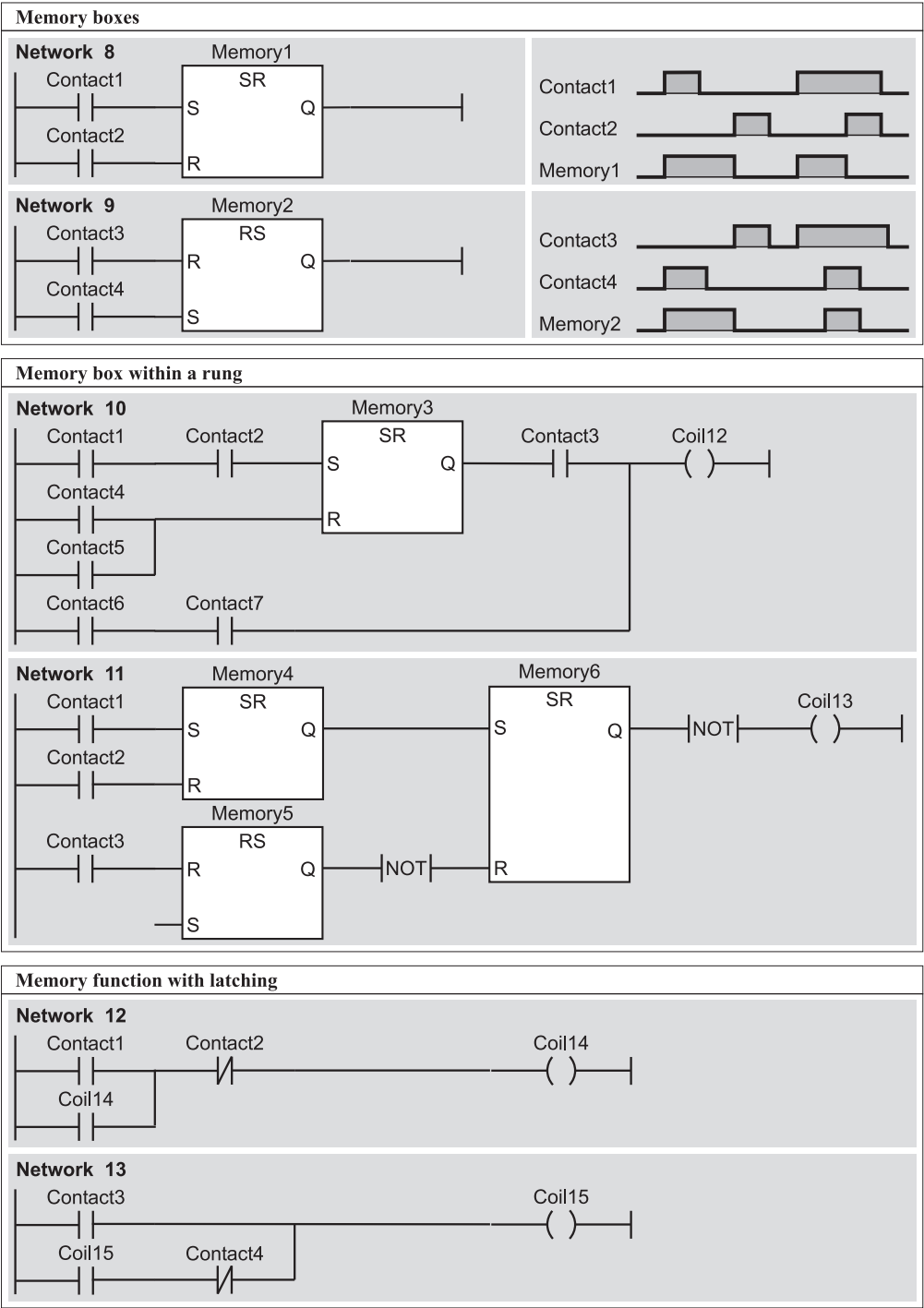


Figure 5.2 Memory Functions (LAD)

output table, and the (external) output on the relevant output module is not affected. The CPU does not transfer the process-image output table to the output modules until the end of the program cycle.

Set priority is the exception when using the memory function. It is used, for example, in the implementation of a fault message buffer if the still current fault message at the set input is to continue to set the memory function despite an acknowledgement at the reset input.

Memory function within a rung

You can also place a memory box within a rung. Contacts can be connected in series and in parallel both at the inputs and at the output (Figure 5.2 Network 10). It is also possible to leave the second input of a memory box unswitched. You can also connect several memory boxes together within one rung. You can arrange the memory boxes in series or in parallel (Network 11).

You can locate a memory function after a T-branch or in a branch that starts at the left power rail.

Memory function with latching

In a relay logic diagram, the memory function is usually implemented by latching the output to be controlled. This method can also be used when programming in ladder logic. However, it has the disadvantage, when compared with the memory box, that the memory function is not immediately recognizable.

Networks 12 and 13 in Figure 5.2 show both types of memory function, set priority and reset priority, using latching. The principle of latching is a simple one. The binary operand controlled with the coil is scanned, and this scan (the “contact of the coil”) is connected in parallel to the set condition. If *Contact1* closes, *Coil14* energizes and closes the contact parallel to *Contact1*. If *Contact1* now opens again, *Coil14* remains energized. *Coil14* de-energizes if *Contact2* opens. If signal state “1” is present at both *Contact1* and *Contact2*, power does not flow into the coil (reset priority). This situation looks different in the lower network: If signal state “1” is present at both *Contact3* and *Contact4*, power flows into the coil (set priority).

5.2 FBD Boxes

In FBD, the memory boxes are used in conjunction with binary logic operations in order to influence the signal states of binary operands with the aid of the result of the logic operation (RLO) generated in the CPU.

Available memory functions are

- ▷ The assign box for dynamic control
- ▷ The boxes set and reset as individually programmed memory functions
- ▷ The boxes RS and SR as full-fledged memory functions
- ▷ The midline output box as intermediate buffer
- ▷ The boxes P and N as edge evaluations of the result of the logic operation
- ▷ The boxes POS and NEG as edge evaluations of operands

The boxes for midline outputs and edge evaluations are discussed in detail in subsequent chapters.

You can use the memory functions described in this chapter in conjunction with all binary operands. There are restrictions when using temporary local data bits as edge memory bits.

The examples shown in this chapter are found in function block FB 105 of the “Basic Functions” program in the “FBD_Book” library that you can download from the publisher's Website (see page 8).

For incremental programming, you will find the program elements for the memory functions in the program element catalog (with VIEW → OVERVIEWS [Ctrl - K] or with INSERT → PROGRAM ELEMENTS) under “Bit Logic”.

5.2.1 Assign

The assign box as terminator of a rung assigns the result of the logic operation directly to the operand adjacent to the box. If the RLO is “1” at the input of the assign box, the binary operand is set; if the RLO is “0”, the operand is reset. The function of the assign box depends on the Master Control Relay (MCR): If the

MCR is activated, signal state “0” is assigned to the binary operand over the box.



A number of examples in Figure 5.3 explain how the assign box works.

Network 1: The operand Output1 directly assumes the signal state of the operand Input1.

Network 2: You can use negation to reverse the function of the assign box.

Network 3: You can direct the RLO to several boxes simultaneously by inserting a T-branch and arranging the boxes with the relevant operands one below the other (“multiple output”). All operands over the boxes respond in the same way.

Network 4: You can insert binary functions between the T-branch and the terminating box, thus expanding a logic operation by additional function boxes.

You will find additional examples for the assign box in Chapter 4.2 “Binary Logic Operations (FBD)”.

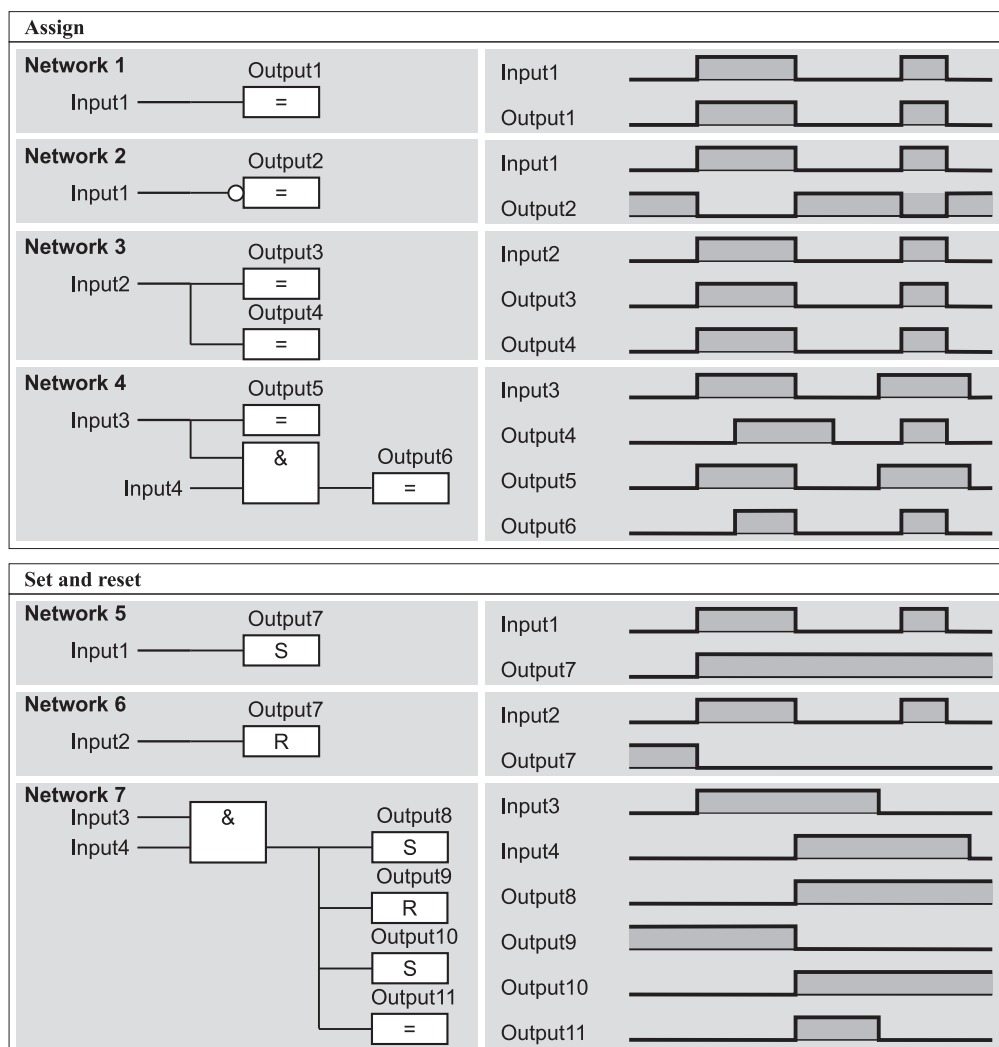


Figure 5.3 Assign, Set and Reset (FBD)

5.2.2 Set and Reset Box

Set and reset boxes also terminate a logic operation. These boxes are activated only when the result of the logic operation going into the box is “1”.



If the RLO going into the set box is “1”, the operand over the box is set to signal state “1”. If the RLO going into the reset box is “1”, the operand over the box is set to signal state “0”. If the RLO going into the set or reset box is “0”, the binary operand remains unaffected. The function of the set and reset boxes depends on the Master Control Relay. If the MCR is activated, the binary operand over the box is not affected.

Figure 5.3 shows several examples of how the set and reset boxes work.

Network 5: The operand *Output7* is set when the operand *Input1* is “1”. *Output7* remains set when *Input1* returns to signal state “0”.

Network 6: The operand *Output7* is reset when the operand *Input2* is “1”. *Output7* remains reset when *Input2* returns to signal state “0”.

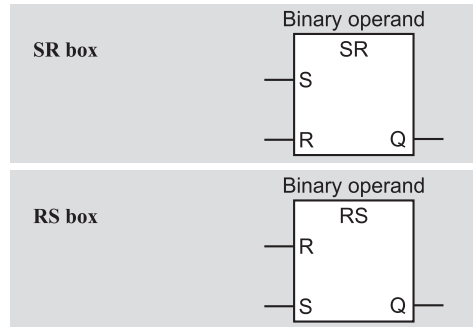
Network 7: You can arrange several set and reset boxes in any combination and together with assign boxes in the same box after a T-branch. As with the assign box, you can also program binary functions after the T-branch and before a set and reset box.

To achieve clarity in your programming, it is advisable to group the set and reset boxes affecting an operand in pairs, and to use them only once in each case. You should also avoid controlling these operands with an assign box.

Please note that the operand used with a set or reset box is usually reset on startup (cold restart or warm restart). In special cases, the signal state is retained. This depends on the startup mode (for instance hot restart), on the operand used (for example static local data), and on the settings in the CPU (such as retentive characteristics).

5.2.3 Memory Box

The functions of the set and reset boxes are summarized in the box of a memory function. The common binary operand is located over the box. Input S of the box corresponds here to the set box, and input R to the reset box. The signal state of the binary operand assigned to the memory function is at output Q of the memory function.



There are two versions of the memory function: As SR box (reset priority) and as RS box (set priority). Apart from the labeling, the boxes also differ from each other in the arrangement of the S and R inputs.

A memory function is set (or, more precisely, the binary operand over the memory box is set) when the set input is “1” and the reset input is “0”. A memory function is reset when the reset input is “1” and the set input is “0”. Signal state “0” at both inputs has no effect on the memory function. If both inputs have signal state “1” simultaneously, the two memory functions respond differently: the SR memory function is reset and the RS memory function is set.

The function of the memory box depends on the Master Control Relay. If the MCR is active, the binary operand of a memory box is no longer affected.

Please note that the operand used with a memory function is normally reset on startup (cold restart or warm restart). In special cases, the signal state of a memory box is retained. This depends on the startup mode (for instance hot restart), on the operand used (for example static local data), and on the settings in the CPU (such as retentive characteristics).

SR memory function

In the SR memory box, the reset input has priority. Reset priority means that the memory function is or remains reset if the RLO is “1” at the set and reset inputs “simultaneously”. The reset input then has priority over the set input (Figure 5.4, Network 8).

Because the statements are executed in sequence, the CPU first sets the memory operand and because the set input is processed first, but resets it again when it processes the reset input. The memory operand remains reset while the rest of the program is processed.

If the memory operand is an output, this brief setting takes place only in the process-image

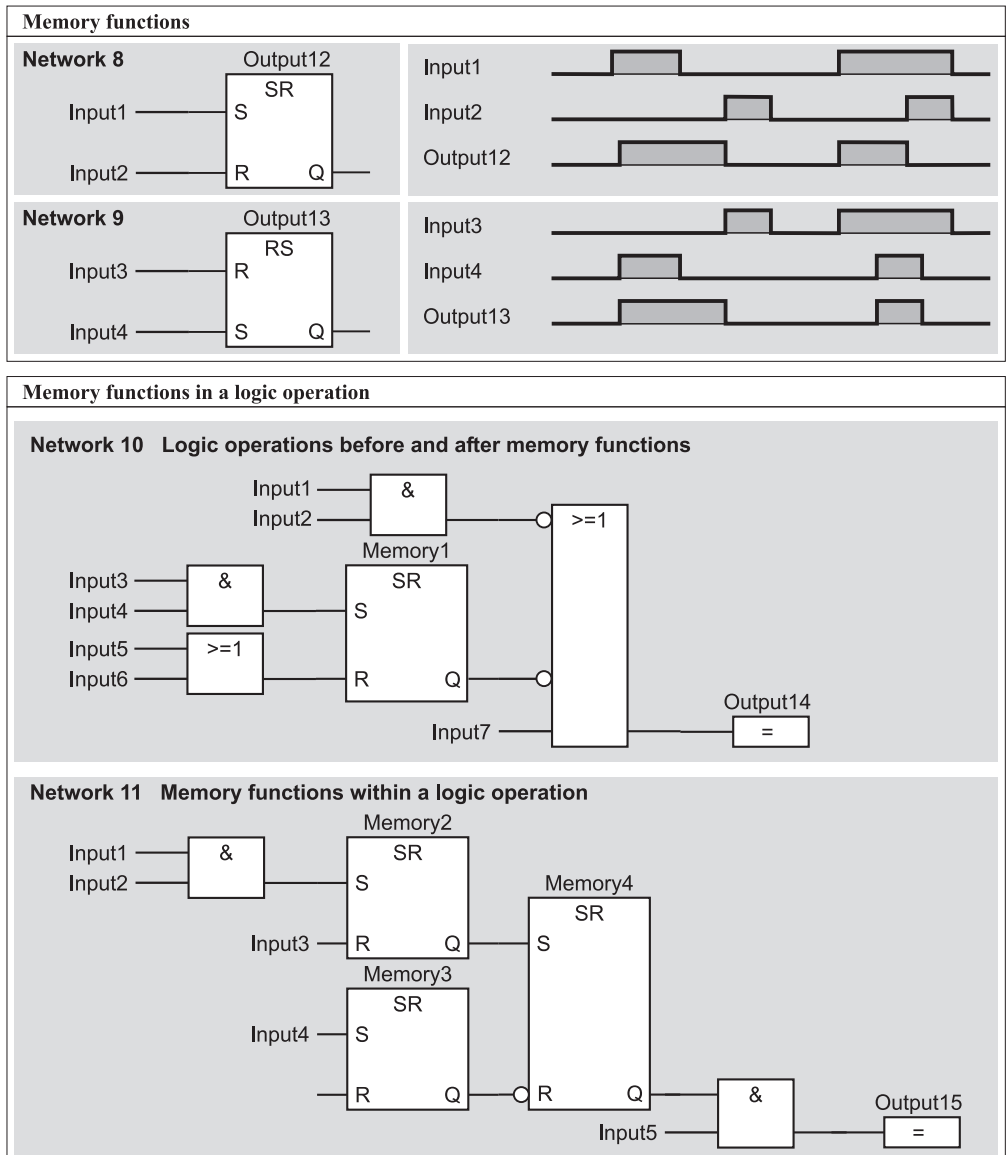


Figure 5.4 Memory Functions (FBD)

output table, and the (external) output on the relevant output module remains unaffected. The CPU does not transfer the process-image output table to the output modules until the end of the program cycle.

The memory function with reset priority is the “normal” form of the memory function, as the reset state (signal state “0”) is usually the safer or less hazardous state.

RS memory function

In the RS memory box, the set input has priority. Set priority means that the memory function is or remains set if the RLO is “1” at the set and reset inputs “simultaneously”. The set input then has priority over the reset input (Figure 5.4, Network 9).

Because the statements are executed in sequence, the CPU initially resets the memory operand because the reset input is processed first, then sets it again when the set input is processed. The memory operand remains set while the rest of the program is processed.

If the memory operand is an output, this brief resetting takes place only in the process-image output table, and the (external) output on the relevant output module is not affected. The CPU does not transfer the process-image output table to the output modules until the end of the program cycle.

Set priority is the exception rather than the rule. Set priority is used, for example, in the implementation of a fault message buffer if the still current fault message at the set input is to continue to set the memory function despite an acknowledgement at the reset input.

Memory function within a logic operation

You can also place a memory box within a logic operation. Binary functions can be programmed both at the inputs and at the output (Figure 5.4, Networks 10 and 11). It is also possible to leave the second input of a memory box unconnected. Within a logic operation, you can also interconnect several memory boxes. The memory boxes may be placed one behind the other or under one another after a T-branch.

5.3 Midline Outputs

Midline outputs are intermediate binary buffers in a ladder diagram or function block diagram. The RLO valid at the midline output is stored in the operand over the midline output. This operand can be scanned again at another point in the program, allowing you to also post-process the RLO valid at midline output elsewhere in the program.

The following binary operands are suitable for intermediate storage of binary results:

- ▷ You can use temporary local data bits if you only require the intermediate result within the block. All code blocks have temporary local data.
- ▷ Static local data bits are available only within a function block; they store the signal state until they are reused, even beyond the block boundaries.
- ▷ Memory bits are available globally in a fixed CPU-specific quantity; for clarity of programming, try to avoid multiple use of memory bits (the same memory bits for different tasks).
- ▷ Data bits in global data blocks are also available throughout the entire program, but before they are used require the relevant data block to be opened (even if implied through mass addressing).

The function of the midline output depends on the Master Control Relay. If the MCR is activated, the binary operand adjacent to the midline output is assigned signal state “0”. The RLO is then “0” following the midline output (that is, there is no longer a “flow of power”).

Note: You can replace the “scratchpad memory” used with STEP 5 by the temporary local data available in every block.

5.3.1 Midline Outputs in LAD

A midline output is a single coil within a rung. The RLO valid up to this point (the power that flows in the rung at this point) is stored in the binary operand over the midline output. The midline output itself has no effect on the power flow.



You can scan the binary operand over the midline output at another point in the program with NO and NC contacts. Several midline outputs can be programmed in one rung.

You can place a midline output in a branch that starts at the left power rail. However, it must not be located directly at the power rail. A midline output may also follow a T-branch, but may not terminate a rung; the single coil is available for this purpose.

Figure 5.5 shows an example of how an intermediate result is stored in a midline output. The RLO from the circuit formed by *Contact1*, *Contact2*, *Contact4* and *Contact5* is stored in

midline output *Midl_out1*. If the condition of the logic operation is fulfilled (power flows in the midline output) and if *Contact3* is closed, *Coil16* is energized. The RLO stored is used in two ways in the next network. On the one hand, a check is made to see if the condition of the logic operation was fulfilled and the bit logic combination made with *Contact6*, and on the other hand a check is made to see if the condition of the logic operation was not fulfilled and a bit combination made with *Contact7*.

5.3.2 Midline Outputs in FBD

A midline output is an assign box within a logic operation. The RLO valid up to this point is stored in the midline output over the midline output box.

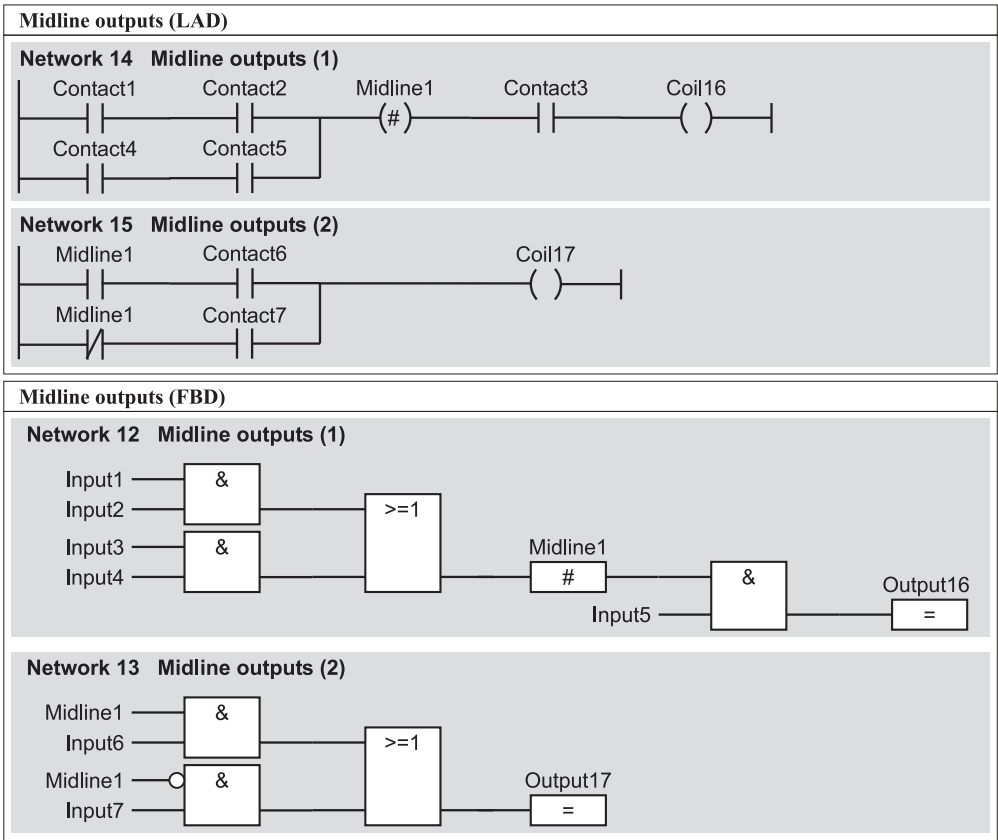


Figure 5.5 Midline Outputs



You can check the binary operand over the midline output at another point in the program. Several midline outputs may be programmed in one logic operation. A midline output box must not terminate a logic operation; the assign box is available for this purpose.

Networks 12 and 13 in Figure 5.5 illustrate how an intermediate result is stored in a midline output. The RLO from the circuit formed by *Input1*, *Input2*, *Input3* and *Input4* is stored in midline output *Midl_out1*. If the condition of the logic operation is fulfilled and if the signal state of *Input5* is “1”, *Output16* is activated. The stored RLO is used in two ways in the next network. On the one hand, a check is made to see if the condition of the logic operation was fulfilled and the bit logic combination made with *Input6*, and on the other hand a check is made to see if the condition of the logic operation was not fulfilled and a bit logic combination made with *Input7*.

5.4 Edge Evaluation

5.4.1 How Edge Evaluation Works

With an edge evaluation, you detect the change in a signal state, a signal edge. An edge is positive (rising) when the signal changes from “0” to “1”. The opposite is referred to as a negative (falling) edge.

In a circuit diagram, the equivalent of an edge evaluation is the pulse contact element. If this pulse contact element emits a pulse when the relay is switched on, this corresponds to the rising edge. A pulse from the pulse contact element on switching off corresponds to a falling edge.

Detection of a signal edge (change in a signal state) is implemented in the program. The CPU compares the current RLO (the result of an input check, for example) with a stored RLO. If the two signal states are different, a signal edge is present.

The stored RLO is located in an “edge memory bit” (it does not necessarily have to be a mem-

ory bit). This must be an operand whose signal state must be available when the edge evaluation is again encountered (in the next program cycle), and which is not used elsewhere in the program. Memory bits, data bits in global data blocks, and static local data bits in function blocks are all suitable as operands.

The edge memory bit stored the “old” RLO with which the CPU last processed the edge evaluation. If a signal edge is now present, that is, if the current RLO differs from the signal state of the edge memory bit, the CPU corrects the signal state of the edge memory bit by assigning it the “value” of the “new” RLO. When the edge evaluation is next processed (usually in the next program cycle), the signal state of the edge memory bit is the same as that of the current RLO (if this has not change in the meantime), and the CPU no longer detects an edge.

A detected edge is indicated by the RLO after edge evaluation. If the CPU detects a signal edge, it sets the RLO to “1” after edge evaluation (power then flows). If there is no signal edge, the RLO is “0”.

Signal state “1” after an edge evaluation therefore means “edge detected”. Signal state “1” is present only briefly, usually only for the length of one program cycle. Since the CPU does not detect an edge in the next cycle (if the “input RLO” of the edge evaluation does not change), it sets the RLO back to “0” after edge evaluation.

Please note the performance characteristics of the edge evaluation when the CPU is switched on. If no edge is to be detected, the RLO prior to edge evaluation must be identical to the signal state of the edge memory bit when the CPU is switched on. Under certain circumstances, the edge memory bit must be reset in the start-up routine (depending on the required performance and on the operand used).

5.4.2 Edge Evaluation in LAD

The LAD programming language provides four different elements for edge evaluation (see Figure 5.6).

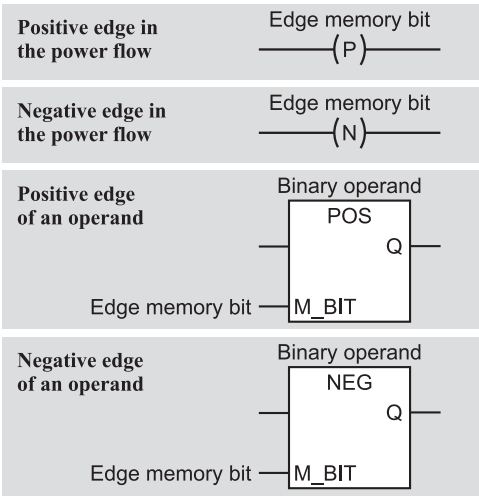


Figure 5.6 LAD elements for edge evaluation

You can process the RLO directly following an edge evaluation, that is to say, store it with a set coil, combine it with downstream contacts, or store it in a binary operand (a so-called “pulse memory bit”). You use a pulse memory bit when the RLO from the edge evaluation is to be used elsewhere in the program; it is, so to speak, the intermediate buffer for a detected edge (the pulse contact element in the circuit diagram). Operands suitable as impulse memory bit are memory bits, data bits in global data blocks, and temporary and static local data bits.

Edge evaluation in the power flow

An edge evaluation in the power flow is indicated by a coil that contains a P (for positive, rising edge) or an N (for negative, falling edge). Above the coil is an edge memory bit, a binary operand, in which the “old” RLO from the preceding edge evaluation is stored. An edge evaluation like this detects a change in the power flow from “power flowing” to “power not flowing” and vice versa.

The example in Figure 5.9 shows a positive and a negative edge evaluation in Network 16. If the parallel circuit consisting of *Contact1* and *Contact3* is fulfilled, the edge evaluation emits a brief pulse with *EmemBit1*. If *Contact2* is closed at this instant, *Memory7* is set. *Memory7* is reset again by a pulse from

EmemBit2 if the series circuit consisting of *Contact4* and *Contact5* interrupts the power flow.

You may program an edge evaluation with coil after a T-branch or in a branch that starts at the left power rail. It must not be placed directly at the left power rail.

Edge evaluation of an operand

LAD represents the edge evaluation of an operand and using a box. Above the box is the operand whose signal state change is to be evaluated. The edge memory bit that stores the “old” signal state from the preceding program cycle is located at input M_BIT.

With the unlabeled input and the output Q, the edge evaluation is “inserted” in the rung instead of a contact. If power flows into the unlabeled input, output Q emits a pulse at an edge; if no power flows in this input, output Q is also always reset. You can arrange this edge evaluation in place of any contact, even in a parallel branch that does not begin at the left power rail.

Figure 5.9 shows the use of an edge evaluation of an operand in Network 17. The edge evaluation in the upper branch emits a pulse if the operand *Contact1* changes its signal state from “0” to “1” (positive edge). This pulse sets *Memory0*. The edge evaluation is always enabled by the direct connection of the unlabeled input to the left power rail. The power edge evaluation is enabled by *Contact2*. If it is enabled with “1” at this input, it emits a pulse if the binary operand *Contact3* changes its signal state from “1” to “0” (negative edge).

5.4.3 Edge Evaluation in FBD

The FBD programming language provides four different elements for edge evaluation (see Figure 5.7).

The RLO after an edge evaluation can be directly processed for example stored with a set box, combined with subsequent binary functions, or assigned to a binary operand (a so-called “pulse memory bit”). A pulse memory bit is used when the RLO from the edge evaluation is to be processed elsewhere in the program; it is, as it were, the intermediate buffer

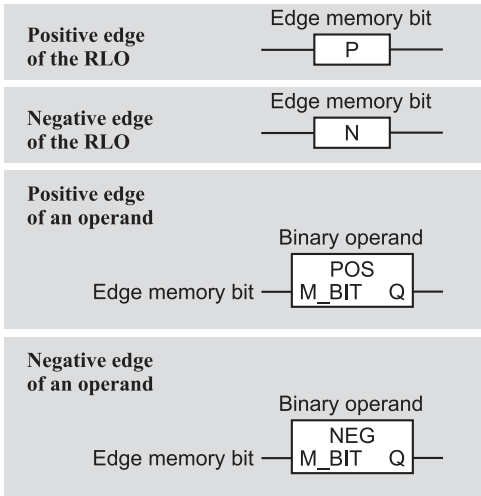


Figure 5.7 FBD elements for edge evaluation

for a detected edge. Operands suitable as pulse memory bits are memory bits, data bits in global data blocks, and temporary and static local data bits.

An edge evaluation is not permitted after a T-branch.

Edge evaluation of the RLO

An edge evaluation of the RLO is indicated by a box that contains a P (for positive, rising edge) or an N (for negative, falling edge). Above the box is the edge memory bit, a binary operand containing the “old” RLO from the previous cycle. An edge evaluation like this detects a change of the RLO within a logic circuit from RLO “1” to RLO “0” and vice versa.

The example in Network 14 of Figure 5.9 shows a positive and a negative edge evaluation. When the OR function consisting of *Input1* and *Input3* is fulfilled, the edge evaluation emits a brief pulse with *EmemBit1*. If *Input2* is “1” at this instant, *Memory1* is set. *Memory1* is reset again by a pulse from *EmemBit2* when the AND function comprising *Input4* and *Input5* is no longer fulfilled.

Edge evaluation of an operand

The edge evaluation of an operand is located at the beginning of a binary logic operation. Over

the box is the operand whose signal state change is to be evaluated. The edge memory bit that holds the “old” signal state from the last program cycle is located at input M_BIT. Output Q is “1” when the CPU detects a signal state change in the operand.

Network 15 in Figure 5.9 edge evaluation of an operand. Upper edge evaluation POS emits a pulse when operand *Input1* goes from “0” to “1” (positive edge). This pulse sets *Memory2*. Lower edge evaluation NEG emits a pulse when binary operand *Input3* goes from “1” to “0” (negative edge). This pulse resets *Memory2* when operand *Input2* is “1”.

5.5 Binary Scaler

A binary scaler has one input and one output. If the signal at the input of the binary scaler changes its state, for example from “0” to “1”, the output also changes its signal state (Figure 5.8). This (new) signal state is then retained until the next signal state change, which is positive in our example. Only then does the signal state of the output change again. This means that half the input frequency appears at the output of the binary scaler.

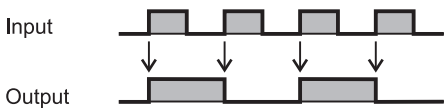


Figure 5.8 Pulse Diagram of a Binary Scaler

5.5.1 Solution in LAD

There are many different ways of solving this task, two of which are presented below.

The first solution uses memory functions (Figure 5.10, Networks 18 and 19). If the signal state of the operand *Input_1* is “1”, the operand *Output_1* is set (the operand *Memory_1* is still reset). If the signal state of the operand *Input_1* changes to “0”, *Memory_1* is also set (*Output_1* is now “1”). If *Input_1* is “1” the next time around, *Output_1* is reset again (*Memory_1* is

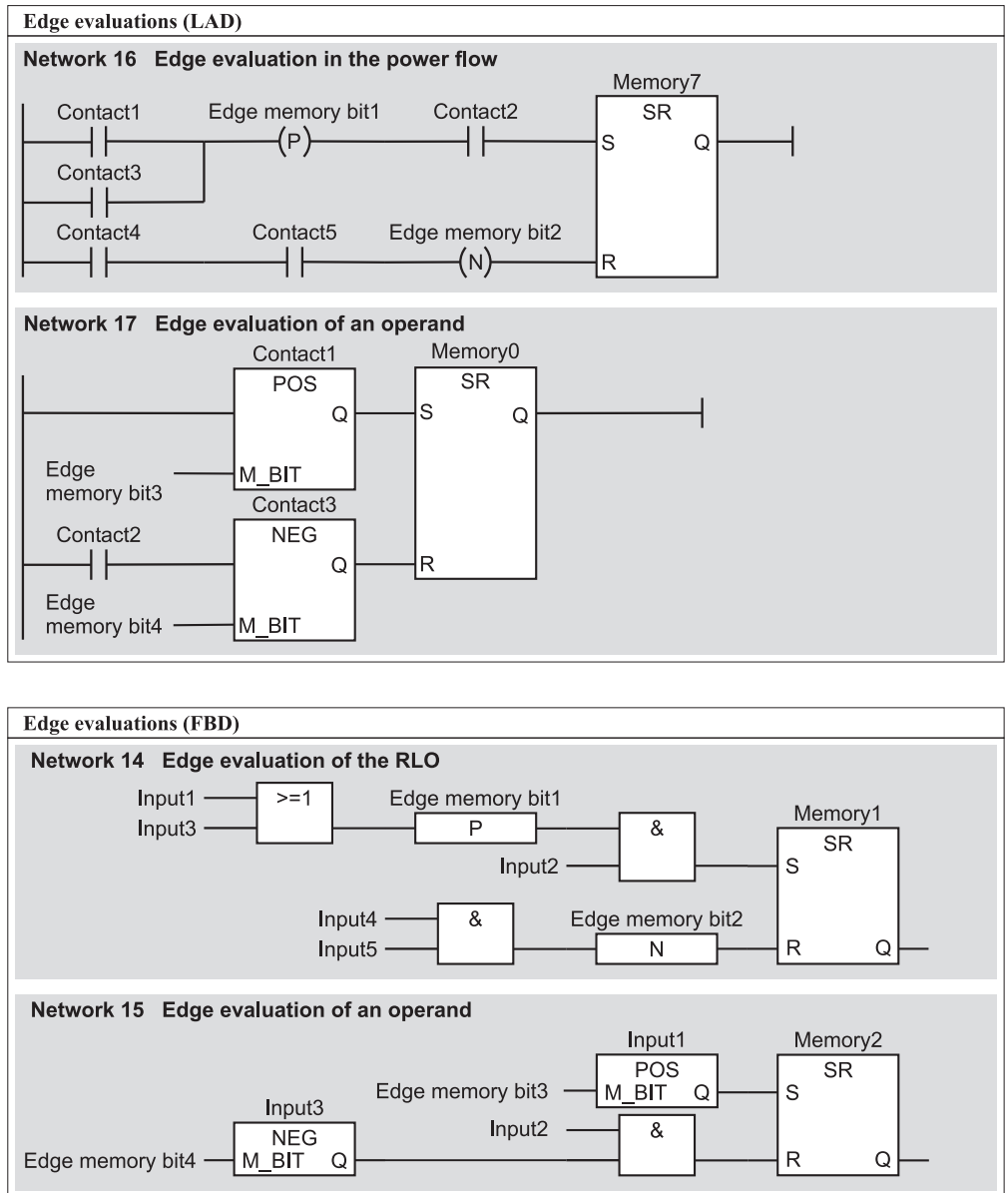


Figure 5.9 Edge Evaluations

now “1”). If *Input_1* is once again “0”, *Memory_1* is reset (since *Output_1* is now also reset). Now the basic state has been reached again after two input pulses and one output pulse.

The second solution uses the latching function (Networks 20 and 21) common in circuit diagrams. The principle is the same as in the first solution except that the reset condition – as is usual with latching – is “zero active”.

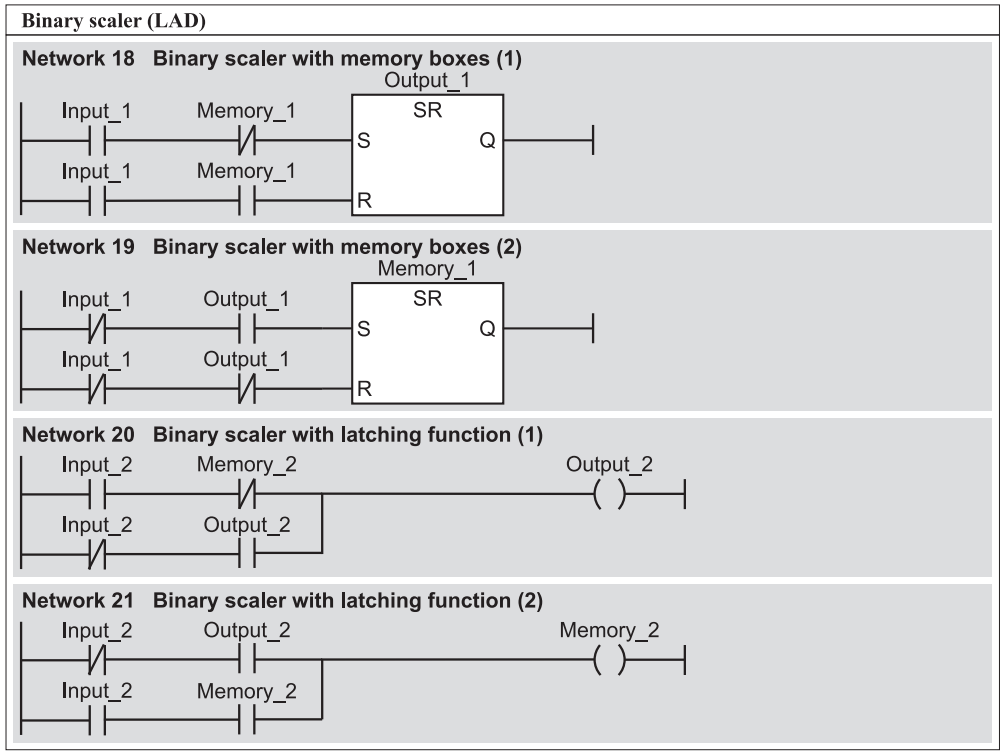


Figure 5.10 Binary Scaler Examples (LAD)

5.5.2 Solution in FBD

There are many different ways of solving this task, two of which are presented below.

The first solution uses memory functions (Figure 5.11, Networks 16 and 17). If the signal state of the operand *Input* is “1”, the operand *Output* is set (the operand *Memory* is still reset). If the signal state of the operand *Input* changes to “0”, *Memory* is also set (*Output* is now “1”). If *Input* is “1” the next time around, *Output* is reset again (*Memory* is now “1”). If *Input* is once again “0”, *Memory* is reset (since *Output* is now also reset). Now the basic state has been reached again after two input pulses and one output pulse.

The second solution uses an edge evaluation of the operand *Input* (Figure 5.11, Networks 18 to 20). If no edge is detected at *Input0*, the RLO is “0” following edge evaluation and the jump

instruction JCN is executed (you will find detailed descriptions of the jump operations in Chapter 16 “Jump Functions”).

In our example, the jump label is called “bin” and is in Network 20. It is here that the program scan is resumed if no edge is detected. The actual binary scaler is in Network 19: If *Output0* is “0”, it is set; if it is “1”, it is reset. This network is processed only when an edge is detected at *Input0*. In essence, every time an edge is detected at *Input0*, *Output0* changes its signal state.

5.6 Example of a Conveyor Control System

The following example of a functionally extremely simple conveyor belt control system illustrates the use of binary logic operations and

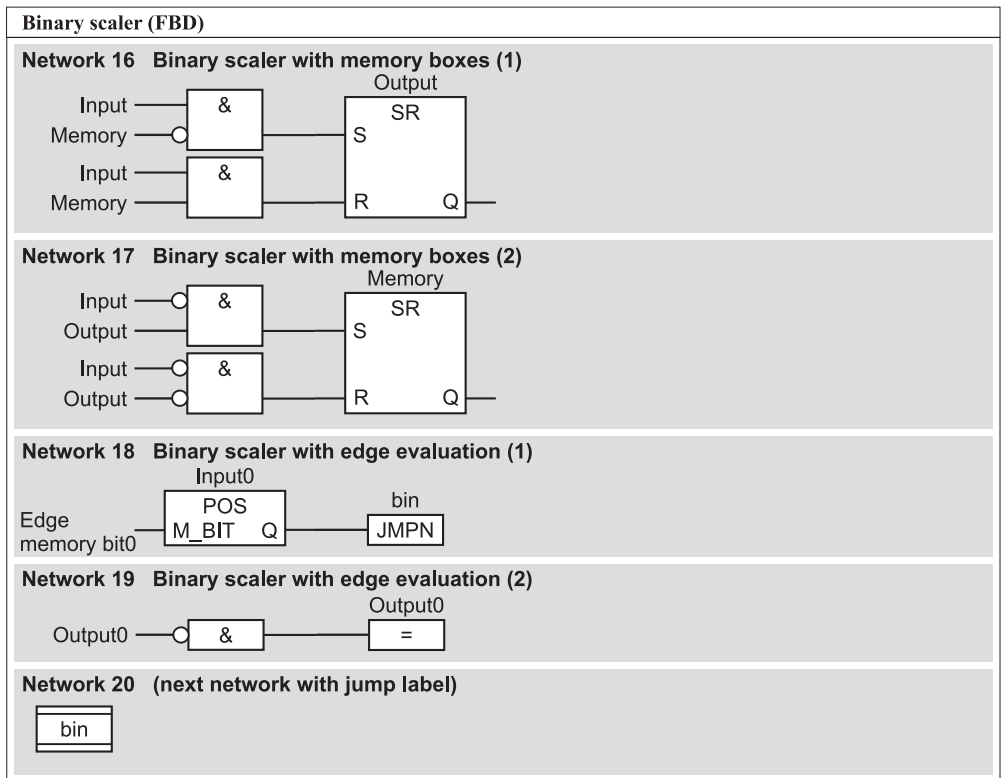


Figure 5.11 Binary Scaler Examples (FBD)

memory functions in conjunction with inputs, outputs and memory bits.

Functional description

Parts are to be transported by conveyor belt, one crate or pallet per belt. The essential functions are as follows:

- ▷ When the belt is empty, the controller requests more parts by issuing the “ready-load” signal (ready to load)
- ▷ When the “Start” signal is issued, the belt starts up and transports the parts
- ▷ At the end of the conveyor belt, an “end-of-belt” sensor (for instance a light barrier) detects the parts, at which point the belt motor switches off and triggers the “ready_rem” signal (ready to remove)
- ▷ When the “continue” signal is issued, the parts are transported further until the “end-

belt” (end-of-belt) sensors no longer detects them.

The example is programmed with inputs, outputs, and memory bits, and may be programmed in any block at any location. In this case, a function without function value was chosen as block.

Signals, symbols

A few additional signals supplement the functionality of the conveyor belt control system:

- ▷ Basic_st
Sets the controller to the basic state
- ▷ Man_on
Switches the belt on, regardless of conditions
- ▷ /Stop
Stops the conveyor as long as the “0” signal

is present (an NC contact as sensor, “zero active”)

- ▷ Light_barrier1
The parts have reached the end of the belt
- ▷ /Mfault1
Fault signal from the belt motor (e.g. motor protection switch); designed as “zero active” signal so that, for example, a wire break also produces a fault signal

We want symbolic addressing, that is, the operands are given names which we then use to write the program. Before entering the program, we create a symbol table (Table 5.1) containing the inputs, outputs, memory bits, and blocks.

Program for LAD

The example is located in a function block that you call in organization block OB 1 (selected from the Program Elements Catalog “FC Blocks”) for processing in a CPU.

Here, the example is programmed with memory boxes. In Chapter 19 “Block Parameters”, the same example is shown using latches. The pro-

gram in this Chapter can be found in a function block with block parameters which can also be called as often as needed (for several conveyor belts).

When programming, the global symbols can also be used without quotation marks provided they do not contain any special characters. If a symbol does contain a special character (such as an umlaut or a space), it must be placed in quotation marks. In the compiled block, the editor indicates all global symbols by setting them in quotation marks.

Figure 5.12 shows the program for the conveyor control system (function block FC 11) under “Conveyor Example” in the “LAD_Book” library that you can download from the publisher's Website (see page 8).

Program for FBD

The example is located in a function which you call in organization block OB 1 (from the Program Elements Catalog “FC Blocks”) for processing in a CPU.

Table 5.1 Symbol Table for the Example “Conveyor Belt Control System”

Symbol	Address	Data Type	Comment
Belt_control	FC 11	FC 11	Belt control system
Basic_st	I 0.0	BOOL	Set controllers to the basic state
Man_on	I 0.1	BOOL	Switch on conveyor belt motor
/Stop	I 0.2	BOOL	Stop conveyor belt motor (zero-active)
Start	I 0.3	BOOL	Start conveyor belt
Continue	I 0.4	BOOL	Acknowledgment that parts were removed
Light_barrier1	I 1.0	BOOL	(Light barrier) sensor signal “End of belt” for belt 1
/Mfault1	I 2.0	BOOL	Motor protection switch belt 1, zero-active
Readyload	Q 4.0	BOOL	Load new parts onto belt (ready to load)
Ready_rem	Q 4.1	BOOL	Remove parts from belt (ready to remove)
Belt_mot1_on	Q 5.0	BOOL	Switch on belt motor for belt 1
Load	M 2.0	BOOL	Load parts command
Remove	M 2.1	BOOL	Remove parts command
EM_Rem_N	M 2.2	BOOL	Edge memory bit for negative edge of “remove”
EM_Rem_P	M 2.3	BOOL	Edge memory bit for positive edge of “remove”
EM_Loa_N	M 2.4	BOOL	Edge memory bit for negative edge of “load”
EM_Loa_P	M 2.5	BOOL	Edge memory bit for positive edge of “load”

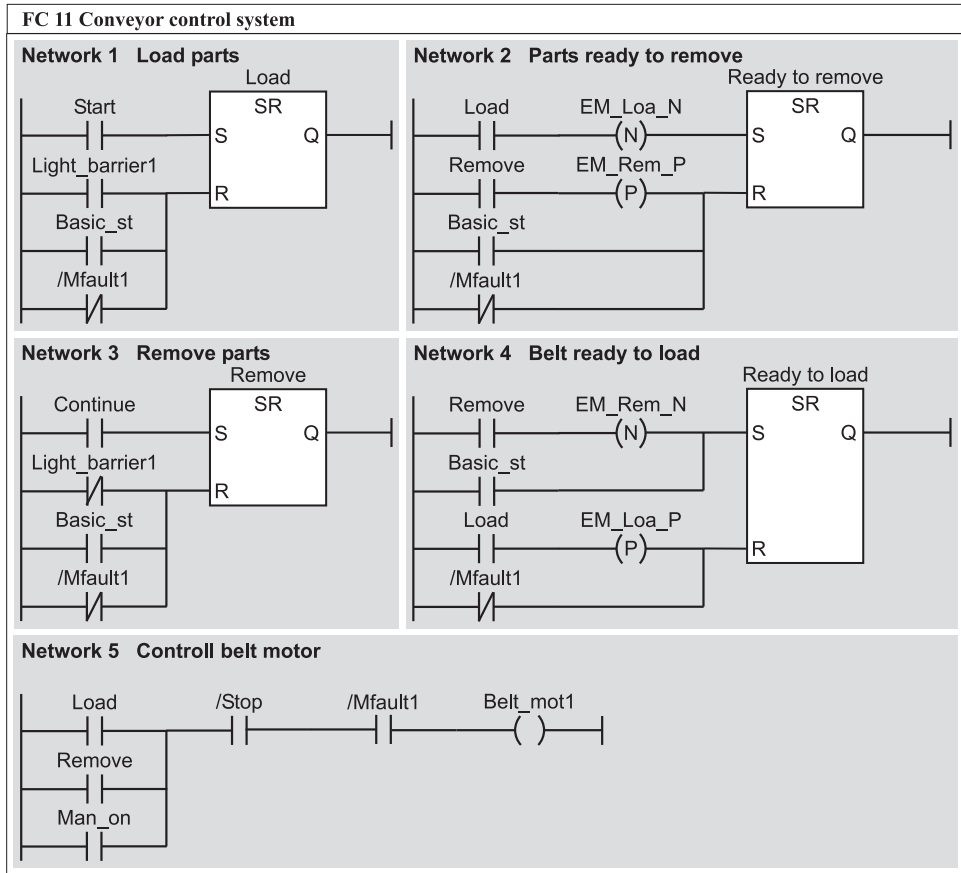


Figure 5.12 Sample Conveyor Control System (LAD)

In Chapter 19 “Block Parameters”, the same example is shown using latches. The program in this Chapter can be found in a function block with block parameters which can also be called as often as needed (for several conveyor belts).

When programming, the global symbols can also be used without quotation marks provided they do not contain any special characters. If a symbol does contain a special character (such

as an umlaut or a space), it must be placed in quotation marks. In the compiled block, the Editor shows all global symbols with quotation marks.

Figure 5.13 shows the circuit diagram for the conveyor control system (function block FC 11) of the “Conveyor Example” program in the “FBD_Book” library that you can download from the publisher’s Website (see page 8).

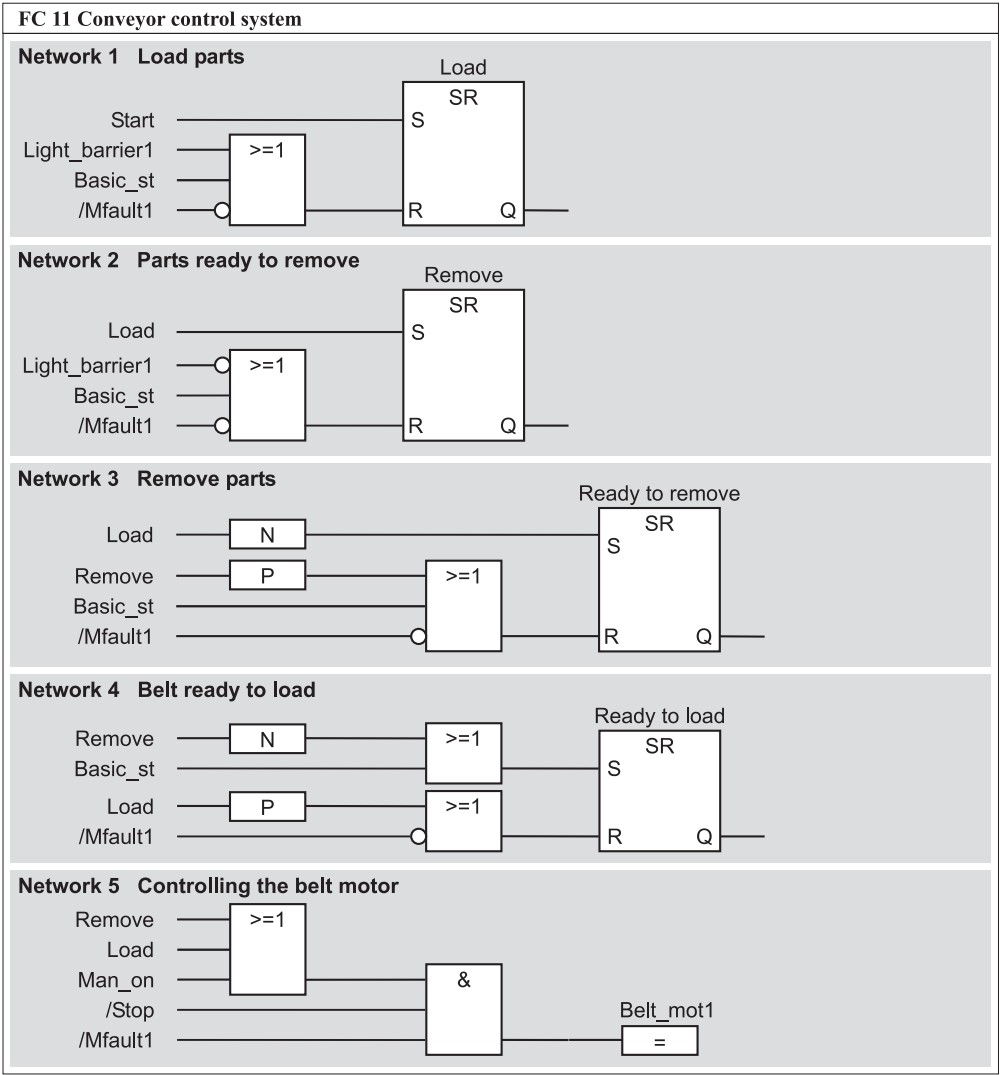


Figure 5.13 Sample Conveyor Control System (FBD)

6 Move Functions

The LAD and FBD programming languages provide the following move functions:

- ▷ MOVE box
Copy operands and variables with elementary data types
- ▷ SFC 20 BLKMOV
Copy data area
- ▷ SFC 21 FILL
Fill data area
- ▷ SFC 81 UBLKMOV
Uninterruptible copying of data area
- ▷ SFC 83 READ_DBL
Read data area from load memory
- ▷ SFC 84 WRIT_DBL
Write data area into load memory

The SFCs are system functions from the standard library *Standard Library* in the *System Function Blocks* program.

6.1 General

You use the move functions to copy information between the system memory, the user memory, and the user data area of the modules (Figure 6.1). Information is transferred via a CPU-internal register that functions as intermediate storage. This register is called accumulator 1. Moving information from memory to accumulator 1 is referred to as “loading” and moving from accumulator 1 to memory is called “transferring”. The MOVE box contains

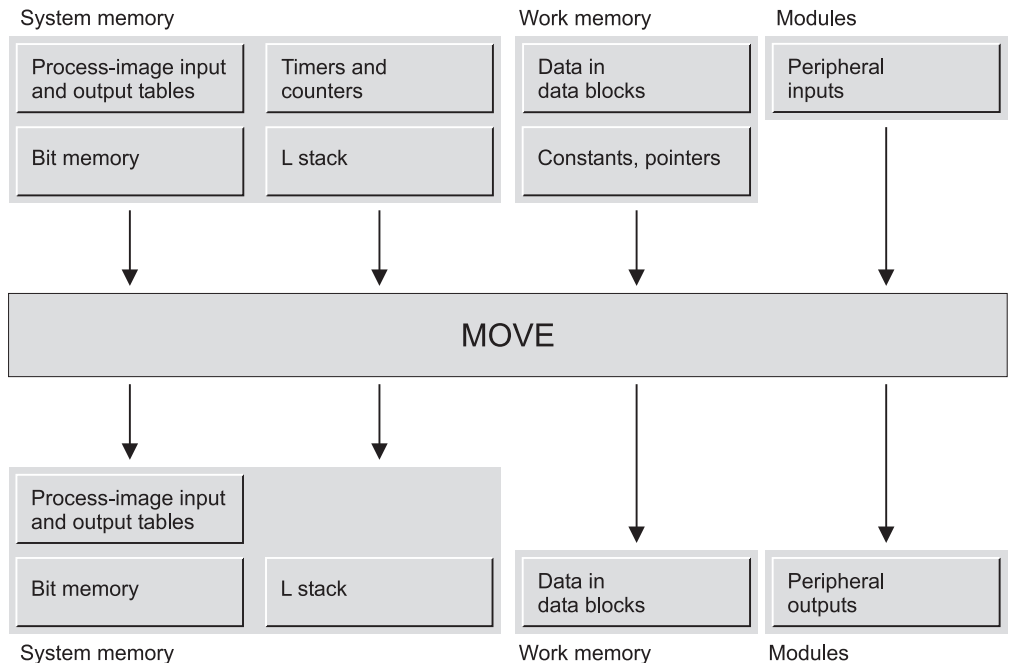


Figure 6.1 Memory Areas for Loading and Transferring

both transfer paths. It moves information at input IN to accumulator 1 (load) and immediately following this from accumulator 1 to the operand at the output (transfer).

6.2 MOVE Box

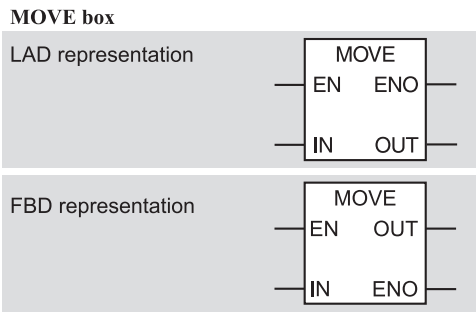
6.2.1 Processing the MOVE Box

Representation

In addition to the enable input EN and the enable output ENO, the MOVE box has an input IN and an output OUT. At the input IN and the output OUT, you can apply all digital operands and digital variables of elementary data type (except BOOL). The variables at input IN and output OUT can have different data types.

The bits in the data formats are discussed in detail in Chapter 3.5 “Variables, Constants and Data Types”.

For incremental programming, you will find the MOVE box in the Program Element Catalog (with VIEW → OVERVIEWS [Ctrl - K] or with INSERT → PROGRAM ELEMENTS) under “Move”.



Different operand widths

The operand widths (byte, word, doubleword) at the input and the output of the MOVE box may vary. If the operand at the input is “less” than at the output, it is moved to the output operand right-justified and is padded at the left with zeroes. If the input operand is “greater” than the output operand, only that part of the input operand on the right that fits into the output operand is moved.

Figure 6.2 explains this. A byte or word at the input is loaded right-justified into accumulator

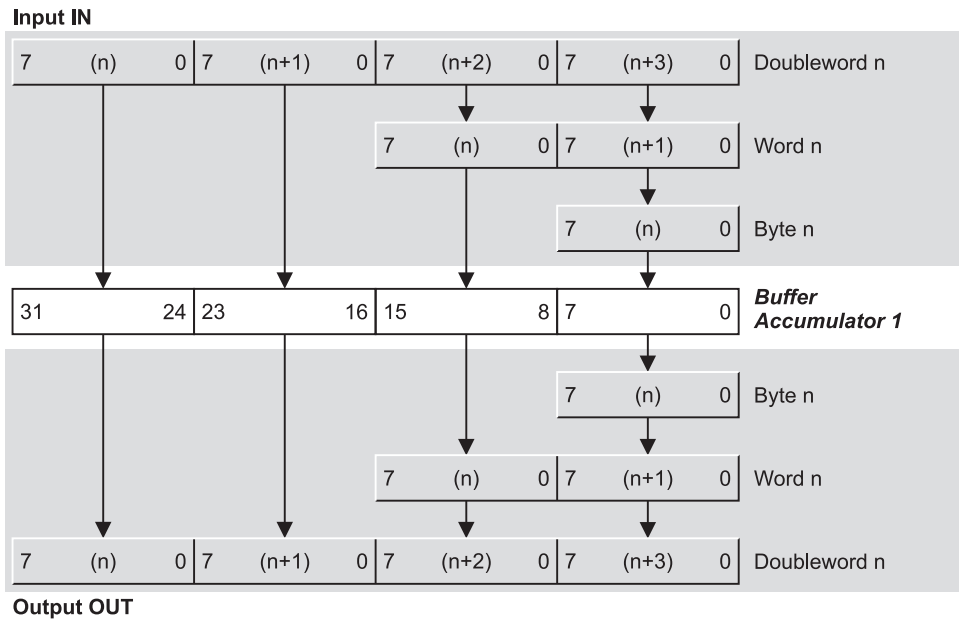


Figure 6.2 Moving Different Operand Widths

1 and the remainder is padded with zeroes. A byte or a word at output OUT is removed right-justified from accumulator 1.

Function

The MOVE box moves the information of the operand at input IN to the operand at output OUT. The MOVE box only moves information when the enable input is “1” or is unused, and when the master control relay is deenergized.

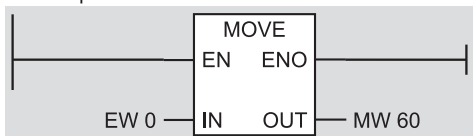
If EN = “1” and the MCR is energized, zero is written to output OUT. With “0” at the enable input, the operand at output OUT is unaffected. MOVE does not report errors.

IF EN == “1” or not wired			ELSE
THEN			
ENO := “1”			
IF MCR enabled			
THEN	ELSE		
OUT := 0	OUT := IN		
			ENO := “0”

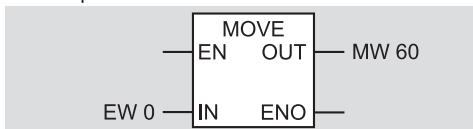
Example

The contents of input word IW 0 are moved to memory word MW 60.

LAD representation:



FBD representation:



MOVE box in a rung (LAD)

You can arrange contacts in series and in parallel before input EN and after output ENO.

The MOVE box must only be placed in a branch that leads directly to the left power rail. This branch can also have contacts before the input EN and it need not be the top branch. With the direct connection to the left power rail you

can connect MOVE boxes in parallel. When connecting boxes in parallel, you require a coil to terminate the rung. If you have not provided for error evaluation, assign a “dummy” operand to the coil, for example a temporary local data bit.

You can connect MOVE boxes in series. In doing so, the ENO output of the preceding box leads to the EN input of the following box.

If you arrange several MOVE boxes in one rung (parallel at the left power rail and then continuing in series), the boxes in the uppermost branch are processed first, from left to right, and then the boxes in the parallel branch from left to right, etc.

You can find examples of the move functions in the “Basic Functions” program (FB 106) of the “LAD_Book” library that you can download from the publisher’s Website (see page 8).

MOVE box in a logic circuit (FBD)

If you want to process the MOVE box in dependence on specific conditions, you can program binary logic circuits before the EN input. You can connect the ENO output with binary inputs of other functions; for instance, you can arrange MOVE boxes in series, whereby the ENO output of the preceding box leads to the EN input of the following box.

EN and ENO need not be wired.

You can find examples of the move functions in the “Basic Functions” program (FB 106) of the “LAD_Book” library that you can download from the publisher’s Website (see page 8).

6.2.2 Moving Operands

In addition to the operands mentioned in this chapter, you can also move timer and counter values (see Chapter 7 “Timers” and Chapter 8 “Counters”). Chapter 18.2 “Block Functions for Data Blocks”, deals with using data operands.

Moving inputs

IB n	Moving an input byte
IW n	Moving an input word
ID n	Moving an input doubleword

Loading from the process image or transferring to the process image of the inputs is also permissible for all CPUs with the S7-300 and for the newer CPUs with the S7-400 in the case of the input bytes which are not present as an input module.

Moving outputs

- QB n Moving an output byte
- QW n Moving an output word
- QD n Moving an output doubleword

Loading from the process image or transferring to the process image of the outputs is also permissible for all CPUs with the S7-300 and for the newer CPUs with the S7-400 in the case of the output bytes which are not present as an output module.

Moving from the I/O

- PIB n Loading a peripheral input byte
- PIW n Loading a peripheral input word
- PID n Loading a peripheral input doubleword
- PQB n Transferring to a peripheral input byte
- PQW n Transferring to a peripheral output word
- PQD n Transferring to a peripheral output doubleword

When moving in the I/O area, you can access different operands depending on the direction of the move. You specify I/O inputs (PIs) at the IN input of the MOVE box, and I/O outputs (PQs) at the OUT output.

When moving from the I/O to memory (loading), the input modules are accessed as peripheral inputs (PIs). Only the available modules may be addressed. Please note that direct loading from the I/O modules can move a different value than loading from the inputs of the module with the same address. While the signal states of the inputs corresponds to the values at the start of the program cycle (when the CPU updated the process image), the values loaded directly from the I/O modules are the current values.

The peripheral outputs (PQs) are used for transfers to the I/O. Only those addresses can be accessed that are also occupied by I/O modules. Transferring to I/O modules that have a process-image output table simultaneously updates that process-image output table, so there is no difference between identically addressed outputs and peripheral outputs.

Moving bit memory

- MB n Moving a memory byte
- MW n Moving a memory word
- MD n Moving a memory doubleword

Moving from and to the bit memory address area is always permissible, since the whole bit memory is in the CPU. Please note here the difference in bit memory area size on the various CPUs.

Moving temporary local data

- LB n Moving a local data byte
- LW n Moving a local data word
- LD n Moving a local data doubleword

Moving from and to the L stack is always allowed. Please note the information in Chapter 18.1.5 “Temporary Local Data”.

6.2.3 Moving Constants

You may specify constant values only at the IN input of the MOVE box.

Moving constants of elementary data type

A fixed value, or constant, can be transferred to an operand. To enhance clarity, this constant can be transferred in one of several different formats. In Chapter 3.5.4 “Elementary Data Types”, you will find an overview of all the different formats. All constants that can be moved using the MOVE box belong to the elementary data types. Examples:

- B#16#F1 Moving a 2-digit hexadecimal number
- 1000 Moving an INT number
- 5.0 Moving a REAL number
- S5T#2s Moving an S5 timer
- TOD#8:30 Moving a time of day

Moving pointers

Pointers are a special form of constant used for calculating addresses in standard blocks. You can use the MOVE box to store these pointers in operands.

P#1.0 Moving an area-internal pointer

P#M2.1 Moving an area-crossing pointer

6.3 System Functions for Data Transfer

The following system functions are available for data transfer

- ▷ SFC 20 BLKMOV
Copy data area
- ▷ SFC 21 FILL
Fill data area
- ▷ SFC 81 UBLKMOV
Uninterruptible copying of a data area
- ▷ SFC 83 READ_DBL
Read from load memory
- ▷ SFC 84 WRIT_DBL
Write into load memory

ANY parameter for the SFCs 20, 21 and 81

These system functions each possess two parameters of data type ANY (Table 6.1). You can connect (in principle) any operand, any variable or any absolute addressed area.

If you use a variable with combined data type, it must only be a “complete” variable; components of a variable (e.g. individual field or structure components) are not permissible. You can use the ANY pointer to define an area with absolute address (see Chapter 6.3.1 “ANY Pointer”).

ANY parameter with the SFC 83 and 84

The system functions SFC 83 READ_DBL and SFC 84 WRIT_DBL transmit data between data blocks present in the load and work memories. Complete data blocks or parts of data blocks are permissible as actual block parameters SRCBLK and DSTBLK. With symbolic addressing, only “complete” variables are accepted which are present in one data block; individual field or structure components are not permissible. Use the ANY pointer to specify an absolute addressed area.

6.3.1 ANY Pointer

You require the ANY pointer when you want to specify an absolute-addressed operand area as block parameter of type ANY. The general format of the ANY pointer is as follows:

P#[DataBlock]Operand Type Quantity

Examples:

P#M16.0 BYTE 8

Area of 8 bytes beginning with MB 16

P#DB11.DBX30.0 INT 12

Area of 12 words in DB 11 beginning with DBB 30

Table 6.1 Parameters for SFC 20, 21 and 81

SFC	Parameter	Declaration	Data Type	Contents, Description
20	SRCBLK	INPUT	ANY	Source area from which data are to be copied
	RET_VAL	RETURN	INT	Error information
	DSTBLK	OUTPUT	ANY	Destination to which data are to be copied
21	BVAL	INPUT	ANY	Source area to be copied
	RET_VAL	RETURN	INT	Error information
	BLK	OUTPUT	ANY	Destination to which the source area is to be copied (including multiple copies)
81	SRCBLK	INPUT	ANY	Source area from which data are to be copied
	RET_VAL	RETURN	INT	Error information
	DSTBLK	OUTPUT	ANY	Destination to which data are to be copied

P#I18.0 WORD 1
Input word IW 18

P#I1.0 BOOL 1
Input I 1.0

Please note that the operand address in the ANY pointer must always be a bit address.

It makes sense to specify a constant ANY pointer when you want to access a data area for which you have not declared variables. In principle, you can assign variables or operands to an ANY parameter. For example, 'P#I1.0 BOOL 1' is identical to 'I 1.0' or the relevant symbolic address.

You can find a more detailed description of the ANY pointer in Chapter 24.2.4 “ANY Pointer”.

6.3.2 Copy Data Area

The system function SFC 20 BLKMOV copies the contents of a source area (parameter SRCBLK) to a destination area (parameter DSTBLK) in the direction of ascending addresses (incremental).

The following actual parameters may be assigned:

- ▷ Any variables from the operand areas for inputs (I), outputs (Q), bit memory (M), and data blocks (variables from global data blocks and from instance data blocks)
- ▷ Variables from the temporary local data (special circumstances govern the use of data type ANY)
- ▷ Absolute-addressed data areas, which require specification of an ANY pointer

You cannot use SFC 20 to copy timers or counters, to copy information from or to the modules (operand area P), or to copy system data blocks (SDBs).

In the case of inputs and outputs, the specified area is copied regardless of whether or not the addresses actually reference input or output modules. If the CPU does not possess SFC 83 READ_DBL, you can also specify a variable or an area from a data block in the load memory as the source area.

Source area and destination area may not overlap. If the source area and the destination area are of different lengths, the transfer is com-

pleted only up to the length of the shorter of the two areas.

Example (Figure 6.3, Network 4): Starting with memory byte MB 64, 16 bytes are to be copied to data block DB 124 starting from DBB 0.

6.3.3 Uninterruptible Copying of a Data Area

System function SFC 81 UBLKMOV copies the contents of a source area (parameter SRCBLK) to a destination area (parameter DSTBLK) in the direction of ascending addresses (incrementally). The copy function is uninterruptible, creating the possibility of increased response times to interrupts. A maximum of 512 bytes can be copied.

The following actual parameters may be assigned:

- ▷ Any variables from the operand areas for inputs (I), outputs (Q), bit memory (M), and data blocks (variables from global data blocks or from instance data blocks)
- ▷ Variables from the temporary local data (special circumstances govern the use of data type ANY)
- ▷ Absolute-addressed data areas, which require specification of an ANY pointer

SFC 81 cannot be used to copy timers or counters, to copy information from or to the modules (operand area P), or to copy system data blocks (SDBs) or data blocks in load memory (data blocks programmed with the keyword *Unlinked*).

In the case of inputs and outputs, the specified area is copied regardless of whether their addresses reference input or output modules.

Source area and destination area may not overlap. If the source area and the destination area are of different lengths, the transfer is completed only up to the length of the shorter of the two areas.

6.3.4 Fill Data Area

System function SFC 21 FILL copies a specified value (source area) to a memory area (destination area) as often as required to fully over-

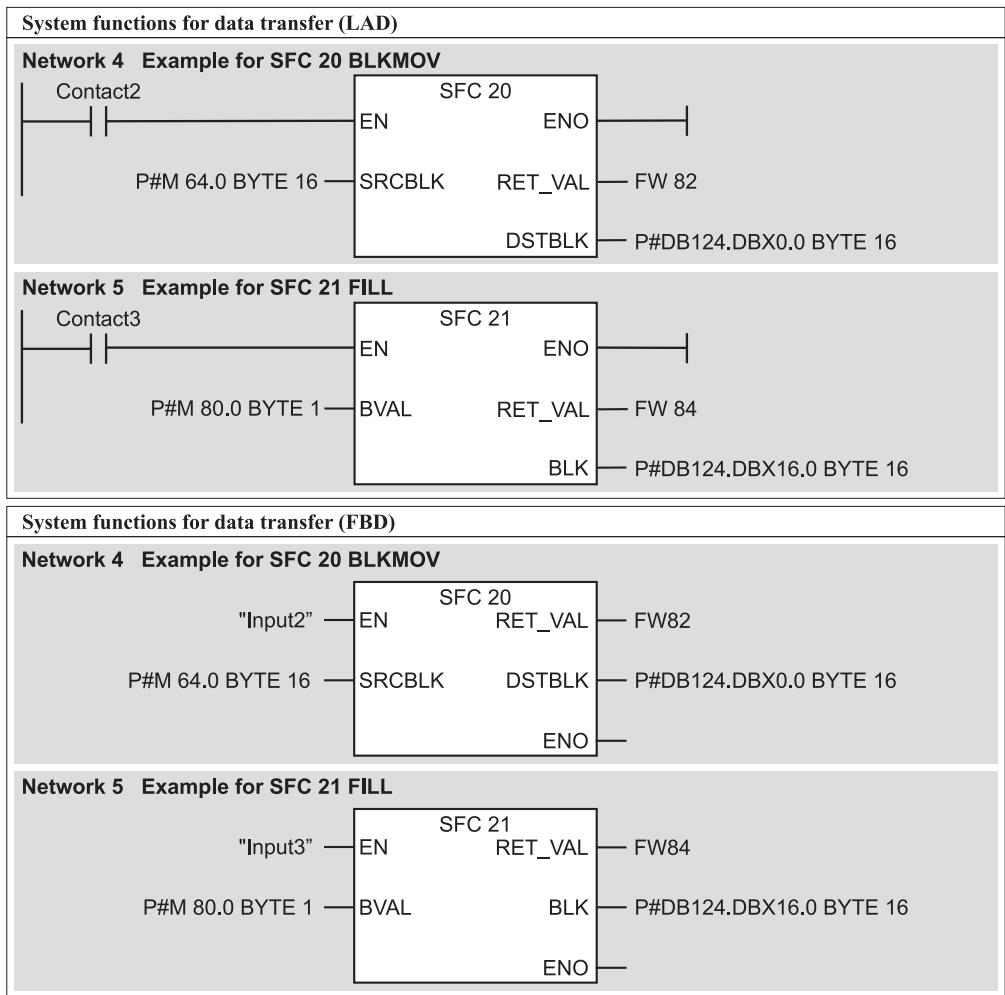


Figure 6.3 Examples for SFC 20 BLKMOV and SFC 21 FILL

write the destination area. The transfer is made in the direction of ascending addresses (incrementally). The following actual parameters may be assigned:

- ▷ Any variables from the operand areas for inputs (I), outputs (Q), bit memory (M), and data blocks (variables from global data blocks and from instance data blocks)
- ▷ Absolute-addressed data areas, requiring specification of an ANY pointer
- ▷ Variables from the temporary local data of data type ANY (special circumstances apply)

SFC 21 cannot be used to copy timers or counters, to copy information from or to the modules (operand area P), or system data blocks (SDBs).

In the case of inputs and outputs, the specified area is copied regardless of whether the addresses actually reference input or output modules.

Source area and destination area may not overlap. The destination area is always fully overwritten, even when the source area is longer than the destination area or when the length of the destination area is not an integer multiple of the length of the source area.

Example (Figure 6.3, Network 5): The contents of memory byte MB 80 is to be copied 16 times to data block DB 124, beginning DBB 16.

6.3.5 Reading from Load Memory

The system function SFC 83 READ_DBL reads data from a data block present in load memory, and writes them into a data block present in work memory. The contents of the read data block are not changed. The block parameters are described in Table 6.2.

The system function SFC 83 READ_DBL operates in asynchronous mode: you trigger the read process with signal state “1” on parameter REQ. You may only access the read and written data areas again when the BUSY parameter has returned to the signal state “0”.

A data block is usually present twice in the user memory of a CPU: once in load memory and – the part relevant to processing – in work memory. If a data block has the attribute *Unlinked*, it is only present in load memory (Figure 6.4). The SFC 83 READ_DBL only reads values from load memory. The initial values of the data operands – which may differ from the actual values in work memory – are present here (see also Chapter 2.6.5 “Block Handling” under “Data blocks offline/online”).

Complete data blocks, e.g. DB 100 or “Recipe 1”, variables from data blocks, or an absolute addressed data area can be specified as ANY

pointers, e.g. P#DB100.DBX16.0 BYTE 64, in the parameters SRCBLK and DSTBLK.

If the source area is smaller than the target area, the source area is written completely into the target area. The remaining bytes of the target area are not changed. If the source area is larger than the target area, the target area is written completely; the remaining bytes of the source area are ignored.

6.3.6 Writing into Load Memory

The system function SFC 84 WRIT_DBL reads data from a data block present in work memory, and writes them into a data block present in load memory. The contents of the read data block are not changed. The block parameters are described in Table 6.2.

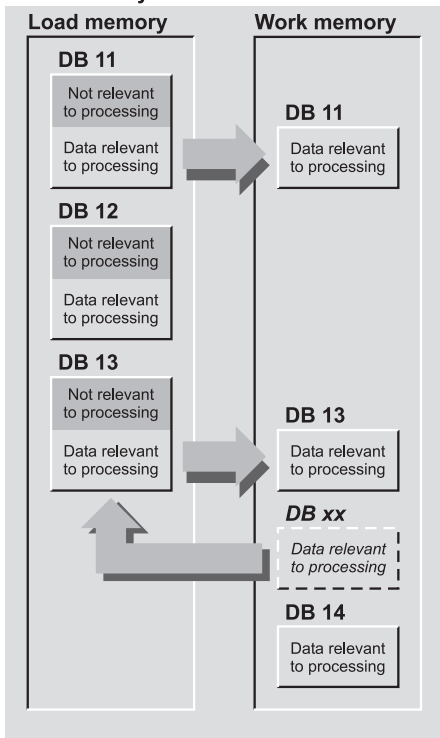
The system function SFC 84 WRIT_DBL operates in asynchronous mode: you trigger the read process with signal state “1” on parameter REQ. You may only access the read and written data areas again when the BUSY parameter has returned to the signal state “0”.

A data block is usually present twice in the user memory of a CPU: once in load memory and – the part relevant to processing – in work memory. If a data block has the attribute *Unlinked*, it is only present in load memory (Figure 6.4). The SFC 84 WRIT_DBL only reads values from work memory. The initial values of the data operands – which may differ from the actual values in load memory – are present here

Table 6.2 Parameters of the SFCs 83 and 84

SFC	Parameter	Declaration	Data Type	Contents, Description
83	REQ	INPUT	BOOL	Trigger reading with signal state “1”
	SRCBLK	INPUT	ANY	Data area in load memory to be read
	RET_VAL	RETURN	INT	Error information
	BUSY	OUTPUT	BOOL	With signal state “1”: reading not yet finished
	DSTBLK	OUTPUT	ANY	Data area in work memory to be written
84	REQ	INPUT	BOOL	Trigger writing with signal state “1”
	SRCBLK	INPUT	ANY	Data area in work memory which is read
	RET_VAL	RETURN	INT	Error information
	BUSY	OUTPUT	BOOL	With signal state “1”: writing not yet finished
	DSTBLK	OUTPUT	ANY	Data area in load memory to be written

User memory of a CPU



A "normal" data block created using the programming device is present twice in the CPU's user memory: it is completely present in load memory, and the data "relative to processing" are present in the work memory, i.e. the data with which the program works.

A data block created using the programming device and with the attribute *Unlinked* is only present in the CPU's load memory. This data block does not occupy any space in work memory.

A data block created using the SFC 82 CREA_DBL is present in load memory and if *Unlinked* is not activated also in work memory. A data block present in work memory is the template according to which the new data block is created.

A data block created using the SFC 22 CREAT_DB or SFC 85 CREA_DB is only present in the CPU's work memory.

Figure 6.4 Data Blocks in User Memory

(see also Chapter 2.6.5 "Block Handling" under "Data blocks offline/online").

Complete data blocks, e.g. DB 200 or "Archive 1", variables from data blocks, or an absolute addressed data area can be specified as ANY pointers, e.g. P#DB200.DBX0.0 WORD 4, in the parameters SRCBLK and DSTBLK.

If the source area is smaller than the target area, the source area is written completely into the target area. The remaining bytes of the target area are not changed. If the source area is larger than

the target area, the target area is written completely; the remaining bytes of the source area are ignored.

Please note: if you write into a data block in load memory (if the initial values are changed), you change the checksum of the user program.

Please also note that the load memory usually only permits a limited number of write operations for physical reasons. Too frequent writing, e.g. cyclic, limits the service life of the load memory.

7 Timers

The timers allow software implementation of timing sequences such as waiting and monitoring times, the measuring of intervals, or the generating of pulses.

The following timer types are available:

- ▷ Pulse timers
- ▷ Extended pulse timers
- ▷ On-delay timers
- ▷ Retentive on-delay timers
- ▷ Off-delay timers

You can program a timer complete as box or using individual program elements. When you start a timer, you specify the type of timer you want it to be and how long it should run; you can also reset a timer. A timer is checked by querying its status (“Timer running”) or the

current time value, which you can fetch from the timer in either binary or BCD code.

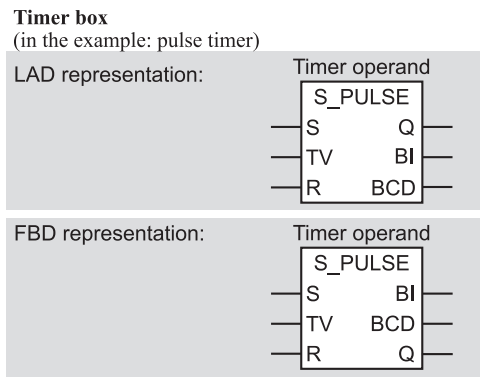
7.1 Programming a Timer

7.1.1 General Representation of a Timer

You can perform the following operations on a timer:

- ▷ Start a timer with specification of the time
- ▷ Reset a timer
- ▷ Check (binary) timer status
- ▷ Check (digital) timer value in binary
- ▷ Check (digital) time value in BCD

The box for a timer contains the coherent representation of all timer operations in the form of function inputs and function outputs (Figure 7.1). Above the box is the absolute or symbolic address of the timer. In the box, as a header, is the timer mode (S_PULSE means “Start pulse timer”). Assignments for the S and TW inputs are mandatory, while assignments for the other inputs and outputs are optional.



Name	Data Type	Description
S	BOOL	Start input
TV	S5TIME	Duration of time specification
R	BOOL	Reset input
BI	WORD	Current time value in binary
BCD	WORD	Current time value in BCD
RP	BOOL	Timer status

Figure 7.1 Timer in Box Representation

Individual program elements in LAD

You can also program a timer using individual program elements (Figure 7.2). The timer is then started via a coil. The timer mode is in the coil (SP = start pulse timer), and below the coil is the value, in S5TIME format, defining the duration. To reset a timer, use the reset coil, and use an NO or NC contact to check the status of the timer. Finally, you can store the current time value, in binary, in a word operand using the MOVE box.

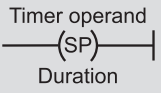
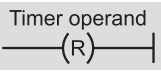
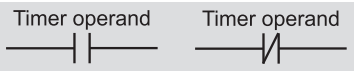
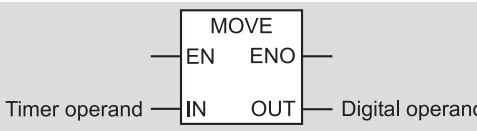
Start timer by specifying time (start coil with time mode)	
Reset timer (reset coil)	
Check timer status (NO contact, NC contact)	
Read time as binary value (MOVE box)	

Figure 7.2 Individual Elements of a Timer (LAD)

Individual program elements in FBD

You can also program a timer using individual program elements (Figure 7.3). The timer is then started via a simple box containing the timer mode (SP = start pulse timer). Below the box is the value, in S5TIME format, defining the duration. To reset a timer, use the reset box. You can scan the status of a timer directly or in negated form with any binary input. Finally, you can store the current time value, in binary, in a word operand using the MOVE box.

For incremental programming, you will find the timers in the Program Element Catalog (with VIEW → OVERVIEWS [Ctrl - K] or INSERT → PROGRAM ELEMENTS) under “Timers”.

7.1.2 Starting a Timer

A timer starts when the result of the logic operation (RLO) changes before the start input or before the start coil/box. Such a signal change is always required to start a timer. In the case of an off-delay timer, the RLO must change from

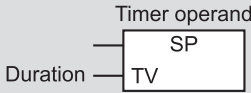
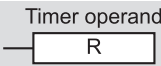
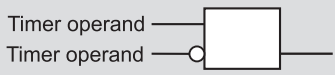
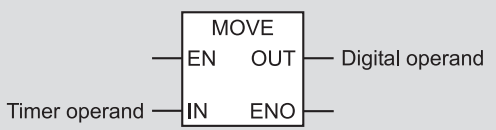
Start timer by specifying time (start coil with time mode)	
Reset timer (reset coil)	
Check timer status (direct or negated binary input)	
Read time as binary value (MOVE box)	

Figure 7.3 Individual Elements of a Timer (FBD)

“1” to “0”; all other timers start when the RLO goes from “0” to “1”.

You can start a timer in one of five different modes (Figure 7.4). There is, however, no point in using any given timer in more than one mode.

7.1.3 Specifying the Duration of Time

The timer adopts the value below the coil/box or the value at input TV as the duration. You can specify the duration as constant, as word operand, or as variable of type S5TIME.

Specifying the duration as constant

S5TIME#10s Duration of 10 s
S5T#1m10ms Duration of 1 min + 10 ms

The duration is specified in hours, minutes, seconds and milliseconds. The range extends from S5TIME#10ms to S5TIME#2h46min30s (which corresponds to 9990 s). Intermediate values are rounded off to 10 ms. You can use S5TIME# or S5T# to identify a constant.

Specifying the duration as operand or variable

MW 20 Word operand containing the duration
“Time1” Variable of data type S5TIME

The value in the word operand must correspond to data type S5TIME (see “Structure of the duration of time value”, below).

Structure of the duration of time value

Internally, the duration is composed of the time value and the time base: duration = time value × time base. The duration is the time during which a timer is active (“timer running”). The time value represents the number of cycles for which the timer is to run. The time base defines the interval at which the CPU is to change the time value (Figure 7.5).

You can also build up a duration of time right in a word operand. The smaller the time base, the more accurate the actual duration. For example, if you want to implement a duration of one second, you can make one of three specifications:

Duration = 2001_{hex} Time base 1 s
Duration = 1010_{hex} Time base 100 ms
Duration = 0100_{hex} Time base 10 ms

The last of these is the preferred one in this case.

When starting a timer, the CPU adopts the programmed time value. The operating system updates the timers at fixed intervals and independently of the user program, that is, it decrements the time value of all active timers as per the time base. When a timer reaches zero, it has run down. The CPU then sets the timer status (signal state “0” or “1”, depending on the mode,

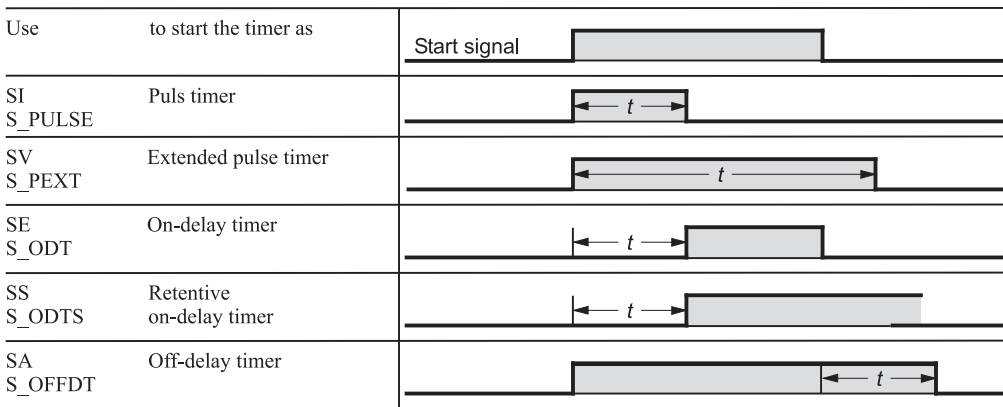


Figure 7.4 Behavioral Characteristics of a Timer

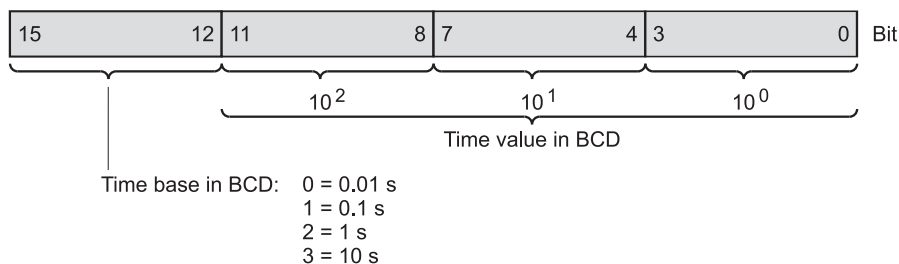


Figure 7.5 Description of the Bits in the Duration

or “type”, of timer) and drops all further activities until the timer is restarted. If you specify a duration of zero (0 ms or W#16#0000) when starting a timer, the timer remains active until the CPU has processed the timer and discovered that the time has elapsed.

Timers are updated asynchronously to the program scan. As a result, it is possible that the time status at the beginning of a cycle is different than at the end of the cycle. If you use the timers at only one point in the program and in the suggested order (see below), the asynchronous updating will prevent the occurrence of malfunctions.

7.1.4 Resetting A Timer

LAD: A timer is reset when power flows in the reset input or in the reset coil (when the RLO is “1”). As long as the timer remains reset, a scan with an NO contact will return “0” and a scan with an NC contact will return “1”.

FBD: A timer is reset when a “1” is present at the reset input. As long as the timer remains reset, a direct scan of the timer status will return “0” and a negated scan will return “1”.

Resetting of a timer sets that timer and the time base to zero. The R input at the timer box need not be wired.

7.1.5 Checking a Timer

Checking the timer status (LAD)

The timer status is found at output Q of the timer box. You can also check the timer status with an NO contact (corresponds to output Q)

or with an NC contact. The results of a check with an NO contact or with output Q differ according to the type of timer (see the description of the timer types, below). As is the case with inputs, for example, a check with an NC contact produces exactly the opposite check result as the one produced by a check with an NO contact. Output Q need not be used at the timer box.

Checking the timer status (FBD)

The timer status is available at output Q of the timer box. You can also check the timer status with a binary function input (corresponding to output Q). The results of a timer check depend on the type of timer involved (see the description of the timer types, below). Output Q need not be used at the timer box.

Checking the time value

Outputs BI and BCD provide the timer's time value in binary (BI) or binary-coded decimal (BCD). It is the value current at the time of the check (if the timer is active, the time value is counted from the set value down towards zero). The value is stored in the specified operand (transfer as with a MOVE box). You do not need to use these outputs at the timer box.

Direct checking of a time value

The time value is available in binary-coded decimal, and can be retrieved in this form from the timer. In so doing, the time base is lost and is replaced with “0”. The value corresponds to a positive number in INT format. Please note: it is the time value that is checked, not the dura-

tion! You can also program direct checking of a time value with the MOVE box.

Coded checking of a time value

You can also retrieve the binary time value in “coded” form from the timer. In this case, both the time value and the time base are available in binary-coded decimal. The BCD value is structured in the same way as for the specification of a time value (see above).

7.1.6 Sequence of Timer Operations

When you program a timer, you do not need to use all the operations available for it. You need use only the operations required to execute a particular function. Normally, these are the operations for starting a timer and for checking the timer status.

In order for a timer to behave as described in this chapter, it is advisable to observe the following order when programming with individual program elements:

- ▷ Start the timer
- ▷ Reset the timer
- ▷ Check the time value or the duration
- ▷ Check the timer status

Omit unnecessary elements when programming. If you observe the order shown above and the timer is started and reset “simultaneously”, the timer will start but will be immedi-

ately reset. The subsequent timer check will fail to detect the fact that the timer was started.

7.1.7 Timer Box in a Rung (LAD)

You can connect contacts in series and in parallel before the start input and the reset input as well as after output Q.

The timer box itself may be located after a T-branch and in a branch that is directly connected to the left power rail. This branch can also have contacts before the start input and it need not be the uppermost branch.

You can find further examples of the representation and arrangement of timers in the “Basic Functions” program (FB 107) of the “LAD_Book” library that you can download from the publisher's Website (see page 8).

7.1.8 Timer Box in a Logic Circuit (FBD)

You can program binary functions and memory functions before the start input and the reset input as well as after output Q.

The timer box and the individual elements for starting and resetting the timer may also be programmed after a T-branch.

You can find further examples for the representation and arrangement of timers in the “Basic Functions” program (FB 107) of the “FBD_Book” library that you can download from the publisher's Website (see page 8).

7.2 Pulse Timer

Starting a pulse timer

The diagram in Figure 7.6 describes the characteristics of a timer when it is started as pulse timer and when it is reset. The description applies if you observe the order shown in Chapter 7.1.6 “Sequence of Timer Operations” when programming with individual elements (starting before resetting before checking).

- ① When the signal state at the timer's start input changes from “0” to “1” (positive edge), the timer is started. It runs for the programmed duration as long as the signal state at the start input is “1”. Output Q supplies signal state “1” as long as the timer runs.

With the start value as the starting point, the time value is counted down toward zero as per the time base.

- ② If the signal state at the timer's start input changes to “0” before the time has elapsed, the timer stops. Output Q then goes to “0”. The time value shows how

much longer the timer would have run had it not been prematurely interrupted.

Resetting a pulse timer

The resetting of a pulse time has a static effect, and takes priority over the starting of a timer (Figure 7.6).

- ③ Signal state “1” at the reset input of an active timer resets that timer. Output Q is then “0”. The time value and the time base are also set to zero. If the signal state at the reset input goes from “1” to “0” while the signal state at the set input is still “1”, the timer remains unaffected.
- ④ Signal state “1” at the reset input of an inactive timer has no effect.
- ⑤ If the signal state at the start input goes from “0” to “1” (positive edge) while the reset signal is still present, the timer starts but is immediately reset (shown by a line in the diagram). If the timer status check was programmed after the reset, the brief starting of the timer does not affect the check.

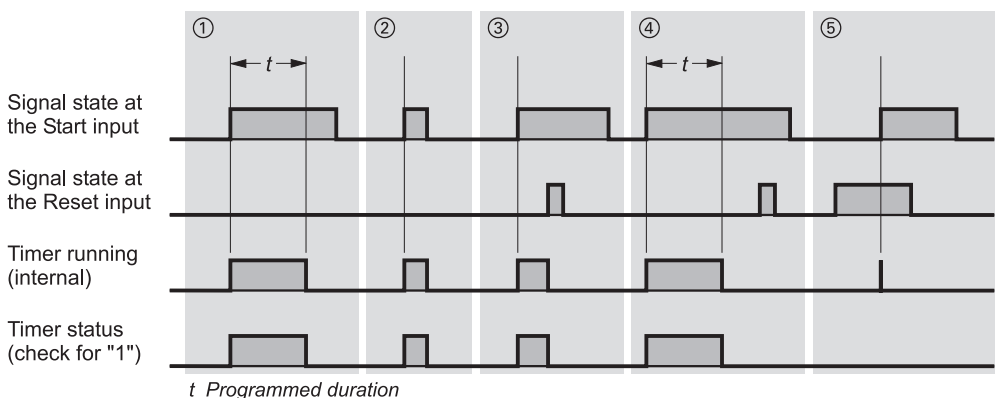


Figure 7.6 Behavioral Characteristics when Starting and Resetting a Pulse Timer

7.3 Extended Pulse Timer

Starting an extended pulse timer

The diagram in Figure 7.7 describes the behavioral characteristics of the timer after it is started and when it is reset. The description applies if you observe the order shown in Chapter 7.1.6 “Sequence of Timer Operations” when programming with individual elements (starting before resetting before checking).

- 1 2** When the signal state at the timer's start input goes from “0” to “1” (positive edge), the timer is started. It runs for the programmed duration, even when the signal state at the start input changes back to “0”. A check for signal state “1” (timer status) returns a check result of “1” as long as the timer is running.

With the start value as starting point, the time value is counted down towards zero as per the time base.

- 3** If the signal state at the start input goes from “0” to “1” (positive edge) while the timer is running, the timer is restarted

with the programmed time value (the timer is “retriggered”). It can be restarted any number of times without first elapsing.

Resetting an extended pulse timer

The resetting of an extended pulse timer has a static effect, and takes priority over the starting of a timer (Figure 7.7).

- 4 5** Signal state “1” at the timer's reset input while the timer is running resets the timer. A check for signal state “1” (timer status) returns a check result of “0” for a reset timer. The time value and the time base are also reset to zero.
- 6** A “1” at the reset input of an inactive timer has no effect.
- 7** If the signal state at the start input goes from “0” to “1” (positive edge) while the reset signal is present, the timer is started but is immediately reset (indicated by a line in the diagram). If the timer status check is programmed after the reset, the brief starting of the timer does not affect the timer check.

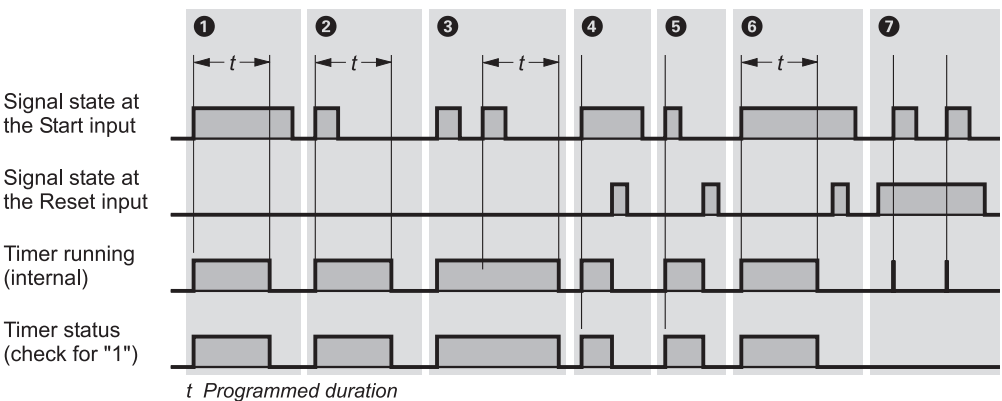


Figure 7.7 Behavioral Characteristics when Starting and Resetting an Extended Pulse Timer

7.4 On-Delay Timer

Starting an on-delay timer

The diagram in Figure 7.8 describes the behavioral characteristics of the timer after it is started and when it is reset. The description applies if you observe the order shown in Chapter 7.1.6 “Sequence of Timer Operations” when programming with individual elements (starting before resetting before checking).

- ① When the signal state at the timer's start input changes from “0” to “1” (positive edge), the timer is started. It runs for the programmed duration. Checks for signal state “1” return a check result of “1” when the time has duly elapsed and signal state “1” is still present at the start input (on-delay).

With the start value as starting point, the time value is counted down towards zero as per the time base.

- ② If the signal state at the start input changes from “1” to “0” while the timer is running, the timer stops. A check for signal state “1” (timer status) always returns a check result of “1” in such cases. The

time value shows the amount of time still remaining.

Resetting an on-delay timer

The resetting of an on-delay timer has a static effect, and takes priority over the starting of the timer (Figure 7.8).

- ③④ Signal state “1” at the reset input resets the timer whether it is running or not. A check for signal state “1” (timer status) then returns a check result of “0”, even when the timer is not running and signal state “1” is still present at the start input. Time value and time base are also set to zero.

A change in the signal state at the reset input from “1” to “0” while signal state “1” is still present at the start input has no effect on the timer.

- ⑤ If the signal state at the start input goes from “0” to “1” (positive edge) while the reset signal is present, the timer starts, but is immediately reset (indicated by a line in the diagram). If the timer status check is programmed after the reset, the brief starting of the timer does not affect the check.

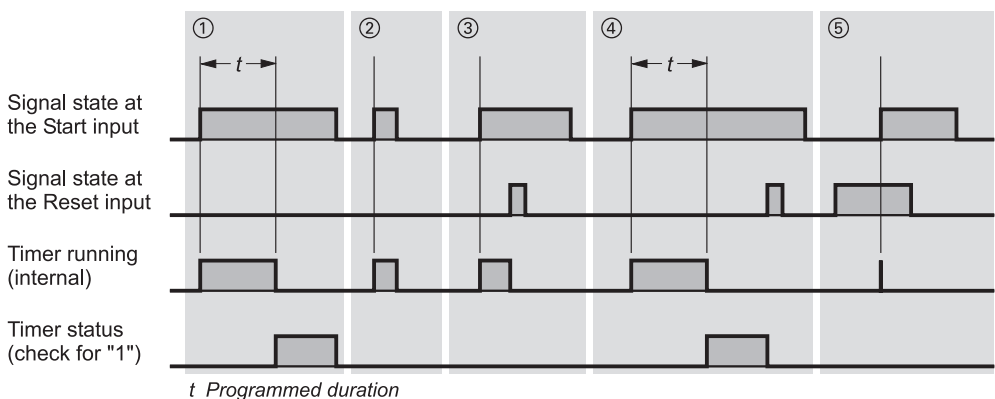


Figure 7.8 Behavioral Characteristics when Starting and Resetting an On-Delay Timer

7.5 Retentive On-Delay Timer

Starting a retentive on-delay timer

The diagram in Figure 7.9 describes the behavioral characteristics of the timer after it is started and when it is reset. The description applies if you observe the order shown in Chapter 7.1.6 “Sequence of Timer Operations” when programming with individual elements (starting before resetting before checking).

1 2 When the signal state at the timer's start input goes from “0” to “1” (positive edge), the timer is started. It runs for the programmed duration, even when the signal state at the start input changes back to “0”. When the time has elapsed, a check for signal state “1” (timer status) returns a check result of “1” regardless of the signal state at the start input. A check result of “0” is not returned until the timer has been reset, regardless of the signal state at the start input. With the start value as starting point, the time value is counted down towards zero as per the time base.

3 If the signal state at the start input changes from “0” to “1” (positive edge) while the timer is running, the timer restarts with the programmed time value (the timer is “retriggered”). It can be restarted any number of times without first having to run down.

Resetting a retentive on-delay timer

The resetting of a retentive on-delay timer has a static effect, and takes priority over the starting of the timer (Figure 7.9).

4 5 Signal state “1” at the reset input resets the timer, regardless of the signal state at the start input. A check for signal state “1” (timer status) then returns a check result of “0”. The time value and the time base are also set to zero.

6 If the signal state at the start input goes from “0” to “1” (positive edge) while the reset signal is present, the timer starts, but is immediately reset (indicated by a line in the diagram). If the timer status check is programmed after the reset, the brief starting of the timer has no effect on the check.

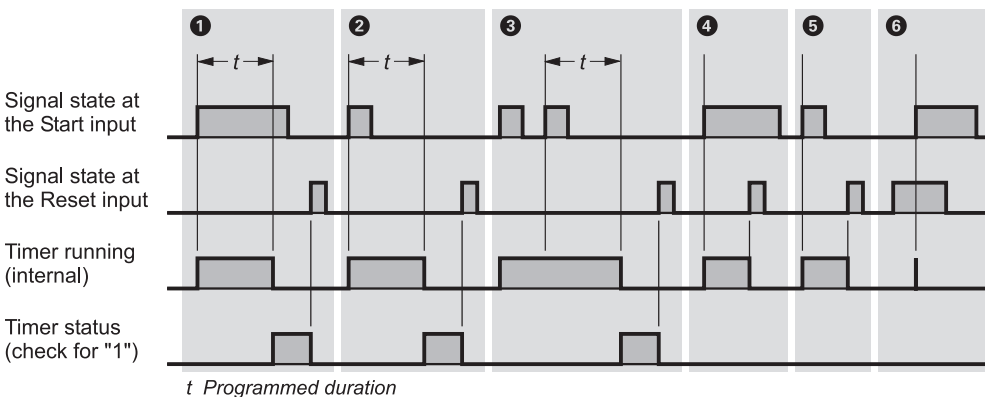


Figure 7.9 Behavioral Characteristics when Starting and Resetting an Retentive On-Delay Timer

7.6 Off-Delay Timer

Starting an off-delay timer

The diagram in Figure 7.10 describes the behavioral characteristics of the timer after it is started and when it is reset. The description applies if you observe the order shown in Chapter 7.1.6 “Sequence of Timer Operations” when programming with individual elements (starting before resetting before checking).

- ①③ The timer starts when the signal state at the timer's start input changes from “1” to “0” (negative edge). It runs for the programmed duration. Checks for signal state “1” (timer status) return a check result of “1” when the signal state at the start input is “1” or when the timer is running (off-delay).

With the start value as starting point, the time value is counted down towards zero as per the time base.

- ② If the signal state at the start input changes from “0” to “1” (positive edge) while the timer is running, the timer is reset. It is re-

started only when there is a negative edge at the start input.

Resetting an off-delay timer

The resetting of an off-delay timer has a static effect, and takes priority over the starting of the timer (Figure 7.10).

- ④ Signal state “1” at the timer's reset input while the timer is running resets the timer. The check result of a check for signal state “1” (timer status) is then “0”. Time value and time base are also set to zero.
- ⑤⑥ Signal state “1” at the start input and at the reset input resets the timer's binary output (a check for signal state “1” (timer status) then returns a check result of “0”). If the signal state at the reset input now changes back to “0”, the timer's output once again goes to “1”.
- ⑦ If the signal state at the start input goes from “1” to “0” (negative edge) while the reset signal is present, the timer starts, but is immediately reset (indicated by a line in the diagram). The check for signal state “1” (timer status) then returns a check result of “0”.

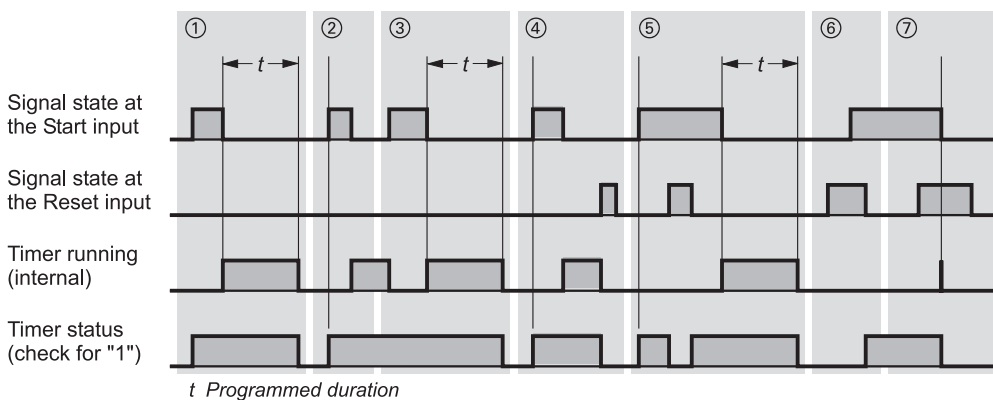


Figure 7.10 Behavioral Characteristics when Starting and Resetting an Off-Delay Timer

7.7 IEC Timers

The IEC timers are integrated in the CPU's operating system as system function blocks (SFBs).

The following timers are available on some CPUs:

- ▷ SFB 3 TP
Pulse timer
- ▷ SFB 4 TON
On-delay timer
- ▷ SFB 5 TOF
Off-delay timer

Figure 7.11 shows the behavioral characteristics of these timers.

These SFBs are called with an instance data block or used as local instances in a function block. You will find the interface description for offline programming in the standard library with the name *Standard Library* under the program *System Function Blocks*.

You will find examples for the call in function block FB 107 of the "Basic Functions" program in the "LAD_Book" and "FBD_Book" libraries that you can download from the publisher's Website (see page 8).

7.7.1 Pulse Timer SFB 3 TP

IEC timer SFB 3 TP has the parameters listed in Table 7.1.

When the RLO at the timer's start input goes from "0" to "1", the timer is started. It runs for the programmed duration, regardless of any subsequent changes in the RLO at the start input. The signal state at output Q is "1" as long as the timer is running.

Table 7.1 Parameters for the IEC Timers

Name	Declaration	Data Type	Description
IN	INPUT	BOOL	Start input
PT	INPUT	TIME	Pulse length or delay duration
RP	OUTPUT	BOOL	Timer status
ET	OUTPUT	TIME	Elapsed time

Output ET supplies the duration of time for output Q. This duration begins at T#0s and ends at the set duration PT. When PT has elapsed, ET remains set to the elapsed time until input IN goes back to "0". If input IN goes to "0" before PT elapses, output ET goes to T#0s the instant PT elapses.

To reinitialize the timer, simply start it with PT = T#0s.

Timer SFB 3 TP is active in START and RUN mode. It is reset (initialized) when a cold start is executed.

7.7.2 On-Delay Timer SFB 4 TON

The IEC timer SFB 4 TON has the parameters listed in Table 7.1.

The timer starts when the RLO at its start input changes from "0" to "1". It runs for the programmed duration. Output Q shows signal state "1" when the time has elapsed and as long as the signal state at the start input remains at "1". If the RLO at the start input changes from "1" to "0" before the time has run out, the timer is reset. The next positive edge restarts the timer.

Output ET supplies the duration of time for the timer. This duration begins at T#0s and ends at set duration PT. When PT has elapsed, ET remains set to the elapsed time until input IN changes back to "0". If input IN goes to "0" before PT elapses, output ET immediately goes to T#0s.

To reinitialize the timer, simply start it with PT = T#0s.

SFB 4 TON is active in START and RUN mode. It is reset on a cold start.

7.7.3 Off-Delay Timer SFB 5 TOF

The IEC timer SFB 5 TOF has the parameters listed in Table 7.1.

The signal state at output Q is "1" when the RLO at the timer's start input changes from "0" to "1". The timer is started when the RLO at the start input changes back to "0". Output Q retains signal state "1" as long as the timer runs. Output Q is reset when the time has elapsed. If the RLO at the start input goes back to "1"

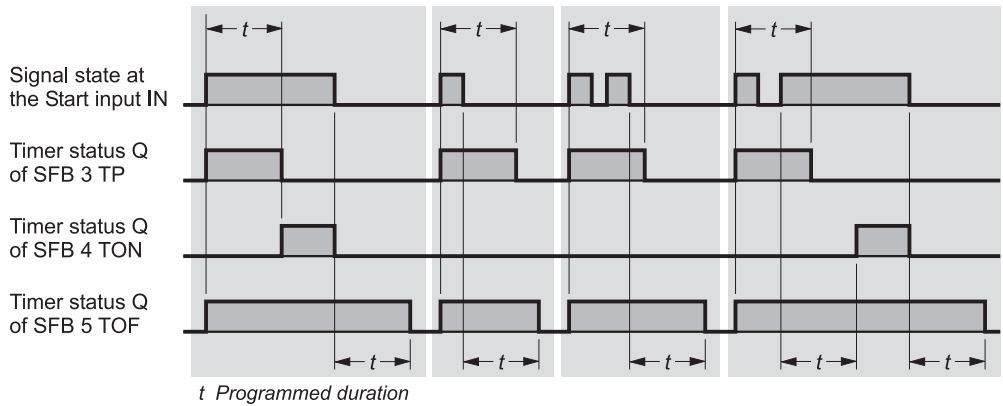


Figure 7.11 Behavioral Characteristics of the IEC Timers

before the time has elapsed, the timer is reset and output Q remains at “1”.

Output ET supplies the duration of time for the timer. This duration begins at T#0s and ends at set duration PT. When PT has elapsed, ET remains set to the elapsed time until input IN changes back to “1”. If input IN goes to “1”

before PT has elapsed, output ET immediately goes to T#0s.

To reinitialize the timer, simply start the timer with PT = T#0s.

SFB 5 TOF is active in START and RUN mode. It is reset on a cold start.

8 Counters

Counters allow you to use the CPU to perform counting tasks. The counters can count both up and down. The counting range extends over three decades (000 to 999). The counters are located in the CPU’s system memory; the number of counters is CPU-specific.

You can program a counter complete as box or using individual program elements. You can set the count to a specific initial value or reset it, and you can count up and down. The counter is scanned by checking the counter status (zero or non-zero count value) or the current count, which you can retrieve in either binary or binary-coded decimal.

8.1 Programming a Counter

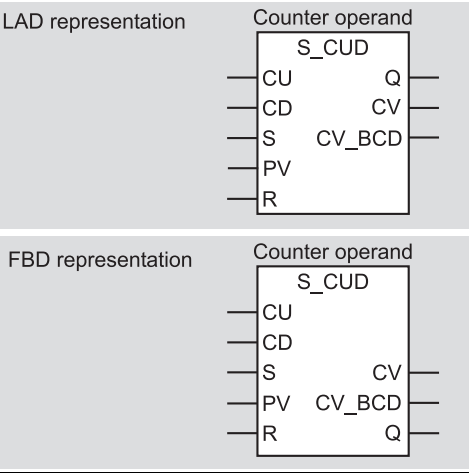
You can perform the following operations on a counter:

- Set counter, specifying the count value
- Count up
- Count down
- Reset counter
- Check (binary) counter status
- Check (digital) count in binary
- Check (digital) count in binary-coded decimal

Representation of a counter as box

A counter box contains the coherent representation of all counting operations in the form of function inputs and function outputs (Figure 8.1). Over the box is the absolute or symbolic address of the counter. In the box, as header, is the counter type (S_CUD stands for “up-down counter”). An assignment is mandatory for the first input (CU in the example) is mandatory;

Counter box
(in the example: up-down counter)



Name	Data Type	Description
CU	BOOL	Up Count input
CD	BOOL	Down Count input
S	BOOL	Set input
PV	WORD	Preset value
R	BOOL	Reset input
CV	WORD	Current value in binary
CV_BCD	WORD	Current value in BCD
RP	BOOL	Counter status

Figure 8.1 Counter in Box Representation

assignments for all other inputs and outputs are optional.

Counter boxes are available in three versions: up-down counter (S_CUD), up counter only (S_CU), and down counter only (S_CD). The differences in functionality are explained below.

For incremental programming, you can find the counters in the Program Element Catalog (with VIEW → OVERVIEWS [Ctrl - K] or INSERT → PROGRAM ELEMENTS) under “Counters”.

Count up function (count up coil)	Counter operand —(CU)—
Count down function (count down coil)	Counter operand —(CD)—
Set counter with value of current count (set counter coil with count)	Counter operand —(SC)— Count value
Reset counter (reset coil)	Counter operand —(R)—
Check counter status (NO contact, NC contact)	Counter operand — Counter operand — /
Read count as binary value (MOVE box)	<div style="text-align: center;"> </div> <div style="display: flex; justify-content: space-between; padding: 0 10px;"> Counter operand — — Digital operand </div>

Figure 8.2 Individual Elements of a Counter (LAD)

Representation of a counter using individual elements (LAD)

You can also program a counter using individual elements (Figure 8.2).

Setting and counting are then done via coils. The set counter coil contains the counting operation (SC = Set Counter); below the coil, in WORD format, is the count value to be used to set the counter.

In the coils for counting, CU stands for count up and CD stands for count down. Use the reset coil to reset a counter and an NO or NC contact to check the status of a counter.

Finally, you can transfer the current count, in binary, with the MOVE box.

Counter box in a rung (LAD)

You can arrange contacts in series and in parallel before the counter inputs, the start input and the reset input as well as after output Q.

The counter box may be placed after a T-branch or in a branch that is directly connected to the left power rail. This branch may also have con-

tacts before the inputs and need not be the uppermost branch.

You can find further examples of the representation and arrangement of counters in the “Basic Functions” program (FB 108) of the “LAD_Book” library that you can download from the publisher's Website (see page 8).

Representation of a counter using individual elements (FBD)

You can also program a counter using individual elements (Figure 8.3).

Setting and counting are then done via simple boxes. The set counter box contains the counting operation (SC = Set Counter); at input PV is the count value, in WORD format, is the count value to be used to set the counter.

In the boxes for counting, CU stands for count up and CD stands for count down. Use the reset box to reset a counter and a direct or negated binary function input to check the status of a counter.

Finally, you can transfer the current count, in binary, with the MOVE box.

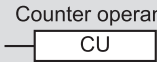
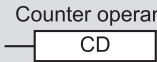
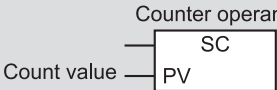
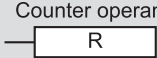
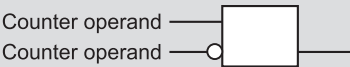
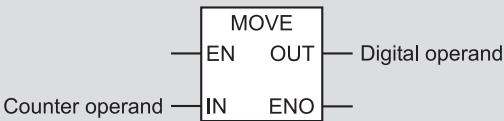
Count up function (count up coil)	
Count down function (count down coil)	
Set counter with value of current count (set counter coil with count)	
Reset counter (reset coil)	
Check counter status (direct or negated binary input)	
Read count as binary value (MOVE box)	

Figure 8.3 Individual Elements of a Counter (FBD)

Counter box in a logic circuit (FBD)

You can arrange binary functions and memory functions before the counter inputs, the start input and the reset input as well as after output Q.

The counter box and the individual elements for counting, setting a counter and resetting a counter may also be placed after a T-branch.

You can find further examples of the representation and arrangement of counters in the “Basic Functions” program (FB 108) of the “FBD_Book” library that you can download from the publisher’s Website (see page 8).

Sequence of counting operations

When programming a counter, you do not need to use all the operations available for that counter. The operations required to carry out the desired functions are enough.

For example, to program a down counter, you need only programs the operations for setting

the counter to its initial count, down counting, and checking the counter status.

In order that a counter’s behavioral characteristics be as described in this chapter, it is advisable to observe the following order when programming with individual program elements:

- ▷ Count (up or down in any order)
- ▷ Set counter
- ▷ Reset counter
- ▷ Check count
- ▷ Check counter status

Omit any individual elements that are not required. If counting, setting, and resetting of the counter take place “simultaneously” when the operations are programmed in the order shown, the count will first be changed accordingly before being reset by the reset operation which follows. The subsequent check will therefore show a count of zero and counter status “0”.

If counting and setting take place “simultaneously” when the operations are programmed in

the order shown, the count will first be changed accordingly before being set to the programmed count value, which it will retain for the remainder of the cycle.

The order of the operations for up and down counting is not significant.

8.2 Setting and Resetting Counters

Setting counters

A counter is set when the RLO changes from “0” to “1” before set input S or before the set coil or set box. A positive edge is always required to set a counter.

“Set counter” means that the counter is set to a starting value. The value may have a range of from 0 to 999.

Specifying the count value

When a counter is set, it assumes the value at input PV or the value below the set coil or set box as count value. You may specify the count value as constant, word operand, or variable of type WORD.

Specifying the count value as constant

C#100	Count value 100
W#16#0100	Count value 100

The count value comprises three decades in the range 000 to 999. Only positive BCD values are permissible; the counter cannot process negative values. You can use C# or W#16# to identify a constant (in conjunction with decimal digits only).

Specifying the count value as operand or variable

MW 56	Word operand containing the count value
“Count value1”	Variable of type WORD

Resetting counters (LAD)

A counter is reset when power flows in the reset input or in the reset coil (when RLO “1” is present). When this is the case, checking the counter with an NO contact will result in a check result of “0”, and checking with an NC contact

will return a check result of “1”. Resetting a counter sets its count to “zero”. The counter box's R input need not be connected.

Resetting counters (FBD)

A counter is reset when a “1” is present at the reset input. Then a direct scan of the counter status will return “0” and a negated scan will return “1”. Resetting a counter sets its count to “zero”. The counter box's R input need not be connected.

8.3 Counting

The counting frequency of a counter is determined by the execution time of your program! To be able to count, the CPU must detect a change in the state of the input pulse, that is, an input pulse (or a space) must be present for at least one program cycle. Thus, the longer the program execution time, the lower the counting frequency.

Up counting

A counter is counted up when the RLO changes from “0” to “1” before the up count input CU or at the up count coil or box. A positive edge is always required for up counting.

In up counting, each positive edge increments the count by one unit until it reaches the upper range limit of 999. Each additional positive edge for up counting then has no further effect. There is no carry.

Down counting

A counter is counted down when the RLO changes from “0” to “1” before down count input CD or at the down count coil or box. A positive edge is always required for down counting.

In down counting, each positive edge decrements the count by one unit until it reaches the lower range limit of 0. Each subsequent positive edge for down counting then has no further effect. There are no negative counts.

Different counter boxes

The Editor provides three different counter boxes:

S_CUD Up/down counter

S_CU Up counter

S_CD Down counter

These counter boxes differ only in the type and number of counter inputs. Whereas S_CUD has inputs for both counting directions, S_CU has only the up count input and S_CD only the down count input.

You must always connect the first input of a counter box. If you do not connect the second input (S_CD) on S_CUD, this box will take on the same characteristics as S_CU.

8.4 Checking a Counter

Checking the counter status (LAD)

The counter status is at output Q of the counter box. You can also check the counter status with an NO contact (corresponding to output Q) or an NC contact.

Output Q is “1” (power flows from the output) when the current count is greater than zero. Output Q is “0” if the current count is equal to zero. Output Q does not need to be connected at the counter box.

Checking the counter status (FBD)

The counter status is at output Q of the counter box. You can also check the counter status directly (corresponds to output Q) with a binary function input, or in negated form.

Output Q is “1” when the current count is greater than zero. Output Q is “0” when the current count is equal to zero. Output Q need not be connected at the counter box.

Checking the count value (LAD and FBD)

Outputs CV and CV_BCD make the counter's current count available in binary (CV) or in binary-coded decimal (BCD). The count value

made available by this operation is the one which is current at the time the check is made.

The value is stored in the specified operand (transfer as with a MOVE box). You need not switch these outputs at the counter box.

Direct checking of the count

The count is available in binary, and can be fetched from the counter in this form. The value corresponds to a positive number in INT format. Direct checking of a count can also be programmed with a MOVE box.

Coded checking of the count

You can also fetch the binary count from the counter in “coded” form. The binary-coded decimal (BCD) value is structured in the same way as for specifying the count (see above).

8.5 IEC Counters

The IEC counters are integrated in the CPU operating system as system function blocks (SFBs). The following counters are available in the appropriate CPUs:

- ▷ SFB 0 CTU
Up counter
- ▷ SFB 1 CTD
Down counter
- ▷ SFB 2 CTUD
Up/down counter

You can call these SFBs with an instance data block or use them as local instances in a function block.

You will find the interface description for offline programming on standard library *Standard Library* under the *System Function Blocks* program.

You will find sample calls in function block FB 108 of the “Basic Functions” program in the “LAD_Book” and “FBD_Book” libraries that you can download from the publisher's Website (see page 8).

8.5.1 Up Counter SFB 0 CTU

IEC counter SFB 0 CTU has the parameters listed in Table 8.1. When the signal state at up count input CU changes from “0” to “1” (positive edge), the current count is incremented by 1 and shown at output CV. On the first call (with signal state “0” at reset input R), the count corresponds to the default value at input PV. When the count reaches the upper limit of 32767, it is no longer incremented, and CU has no effect.

The count value is reset to zero when the signal state at reset input R is “1”. As long as input R is “1”, a positive edge at CU has no effect. Output Q is “1” when the value at CV is greater than or equal to the value at PV.

SFB 0 CTU executes in START and RUN mode. It is reset on a cold start.

8.5.2 Down Counter SFB 1 CTD

IEC counter SFB 1 CTD has the parameters listed in Table 8.1. When the signal state at down count input CD goes from “0” to “1” (positive edge), the current count is decremented by 1 and shown at output CV. On the first call (with signal state “0” at load input LOAD), the count corresponds to the default value at input PV. When the current count reaches the lower limit of –32768, it is no longer decremented and CD has no effect.

The count is set to default value PV when load input LOAD is “1”. As long as input LOAD is “1”, a positive edge at input CD has no effect.

Output Q is “1” when the value at CV is less than or equal to zero.

SFB 1 CTD executes in START and RUN mode. It is reset on a cold start.

8.5.3 Up/down Counter SFB 2 CTUD

IEC counter SFB 2 CTUD has the parameters listed in Table 8.1.

When the signal state at up count input CU changes from “0” to “1” (positive edge), the count is incremented by 1 and shown at output CV. If the signal state at down count input CD changes from “0” to “1” (positive edge), the count is decremented by 1 and shown at output CV. If both inputs show a positive edge, the current count remains unchanged.

If the current count reaches the upper limit of 32767, it is no longer incremented when there is a positive edge at count up input CU. CU then has no further effect. If the current count reaches the lower limit of –32768, it is no longer decremented when there is a positive edge at down count input CD. CD then has no effect.

The count is set to default value PV when load input LOAD is “1”. As long as load input LOAD is “1”, positive signal edges at the two count inputs have no effect.

Table 8.1 Parameters for the IEC Counters

Name	Present in SFB			Declaration	Data Type	Description
CU	0	-	2	INPUT	BOOL	Up count input
CD	-	1	2	INPUT	BOOL	Down Count input
R	0	-	2	INPUT	BOOL	Reset input
LOAD	-	1	2	INPUT	BOOL	Load input
PV	0	1	2	INPUT	INT	Preset value
Q	0	1	-	OUTPUT	BOOL	Counter status
QU	-	-	2	OUTPUT	BOOL	Up counter status
QD	-	-	2	OUTPUT	BOOL	Down counter status
CV	0	1	2	OUTPUT	INT	Current count value

The count is reset to zero when reset input R is “1”. As long as input R is “1”, positive signal edges at the two count inputs and signal state “1” at load input LOAD have no effect.

Output QU is “1” when the value at CV is greater than or equal to the value at PV.

Output QD is “1” when the value at CV is less than or equal to zero.

SFB 2 CTUD executes in START and RUN mode. It is reset on a cold start.

8.6 Parts Counter Example

The examples illustrates the use of timers and counters. It is programmed with inputs, outputs and memory bits so that it can be programmed at any point in any block. At this point, a function without block parameters is used; the timers and counters are represented by complete boxes. You will find the same example programmed as a function block with block parameters and with individual elements in Chapter 19 “Block Parameters”.

Function description

Parts are transported on a conveyor belt. A light barrier detects and counts the parts. After a set number, the counter sends the *Finished* signal. The counter is equipped with a monitoring circuit. If the signal state of the light barrier does not change within a specified amount of time, the monitor sends a signal.

The *Set* input passes the starting value (the number to be counted) to the counter. A positive edge at the light barrier decrements the counter by one unit. When a value of zero is reached, the counters sends the *Finished* signal. Prerequisite is that the parts be arranged singly (at intervals) on the belt.

The *Set* input also sets the *Active* signal. The controller monitors a signal state change at the light barrier in the active state only. When counting is finished and the last counted item has exited the light barrier, *Active* is reset.

In the active state, a positive edge at the light barrier starts the timer with the time value *Duration1* (“Dura1”) as retentive pulse timer. If the timer's start input is processed with “0” in the next cycle, it still continues to run. A new positive edge “retriggers” the timer, that is,

Table 8.2 Symbol Table for the Parts Counter Example

Symbol	Address	Data Type	Comment
Counter_control	FC 12	FC 12	Counter and monitor control for parts
Acknowl	I 0.6	BOOL	Acknowledge fault
Set	I 0.7	BOOL	Set counter, activate monitor
Lbarr1	I 1.0	BOOL	“End_of_belt” sensor signal conveyor belt 1
Finished	Q 4.2	BOOL	Number of parts reached
Fault	Q 4.3	BOOL	Monitor responded
Active	M 3.0	BOOL	Counter and monitor active
EM_LB_P	M 3.1	BOOL	Edge memory bit for positive edge of light barrier
EM_LB_N	M 3.2	BOOL	Edge memory bit for negative edge of light barrier
EM_Ac_P	M 3.3	BOOL	Edge memory bit for positive edge of “Monitor active”
EM_ST_P	M 3.4	BOOL	Edge memory bit for positive edge of “Set”
Quantity	MW 4	WORD	Number of parts
Dura1	MW 6	S5TIME	Monitoring time for light barrier covered
Dura2	MW 8	S5TIME	Monitoring time for light barrier not covered
Count	C 1	COUNTER	Counter function for parts
Monitor	T 1	TIMER	Timer function for monitor

restarts it. The next positive edge to restart the timer is generated when the light barrier signals a negative edge. The timer is then started with the time value *Duration2* (“Dura2”). If the light barrier is now covered for a period of time exceeding *Dura1* or free for a period of time exceeding *Dura2*, the timer runs down and signals *Fault*. The first time it is activated, the timer is started with the time value *Dura2*.

Signals, symbols

The *Set* signal activates the counter and the monitor. The light barrier controls the counter, the *active* state, selection of the time value, and the starting (retriggering) of the monitoring time via positive and negative edges.

Evaluation of the positive and negative edge of the light barrier is required often, and temporary local data are suitable as “scratchpad” memory. Temporary local data are block-local variables; they are declared in the blocks (not in the symbol table). In the example, the pulse memory bits used for edge evaluation are stored in temporary local data. (The edge memory bits require their signal states in the next cycle as well, and must therefore not be temporary local data.)

We want symbolic addressing, that is, the operands are assigned names which are then used for programming. Before entering the program, we create a symbol table (Table 8.2) containing the inputs, outputs, memory bits, timers, counters, and blocks.

Program

The program is located in a function that you call in the CPU in organization block OB 1 (selected from the Program Elements Catalog under “FC Blocks”).

During programming, the global symbols can also be used without quotation marks provided they do not contain any special characters. If a symbol contains a special character (an Umlaut or a space, for instance), it must be placed in quotation marks. In the compiled block, the Editor shows all global symbols with quotation marks

Figure 8.4 and Figure 8.5 shows the program for the parts counter (Function FC 12). You can find this program in the “Conveyor Example” program of the “LAD_Book” and “FBD_Book” libraries that you can download from the publisher's Website (see page 8).

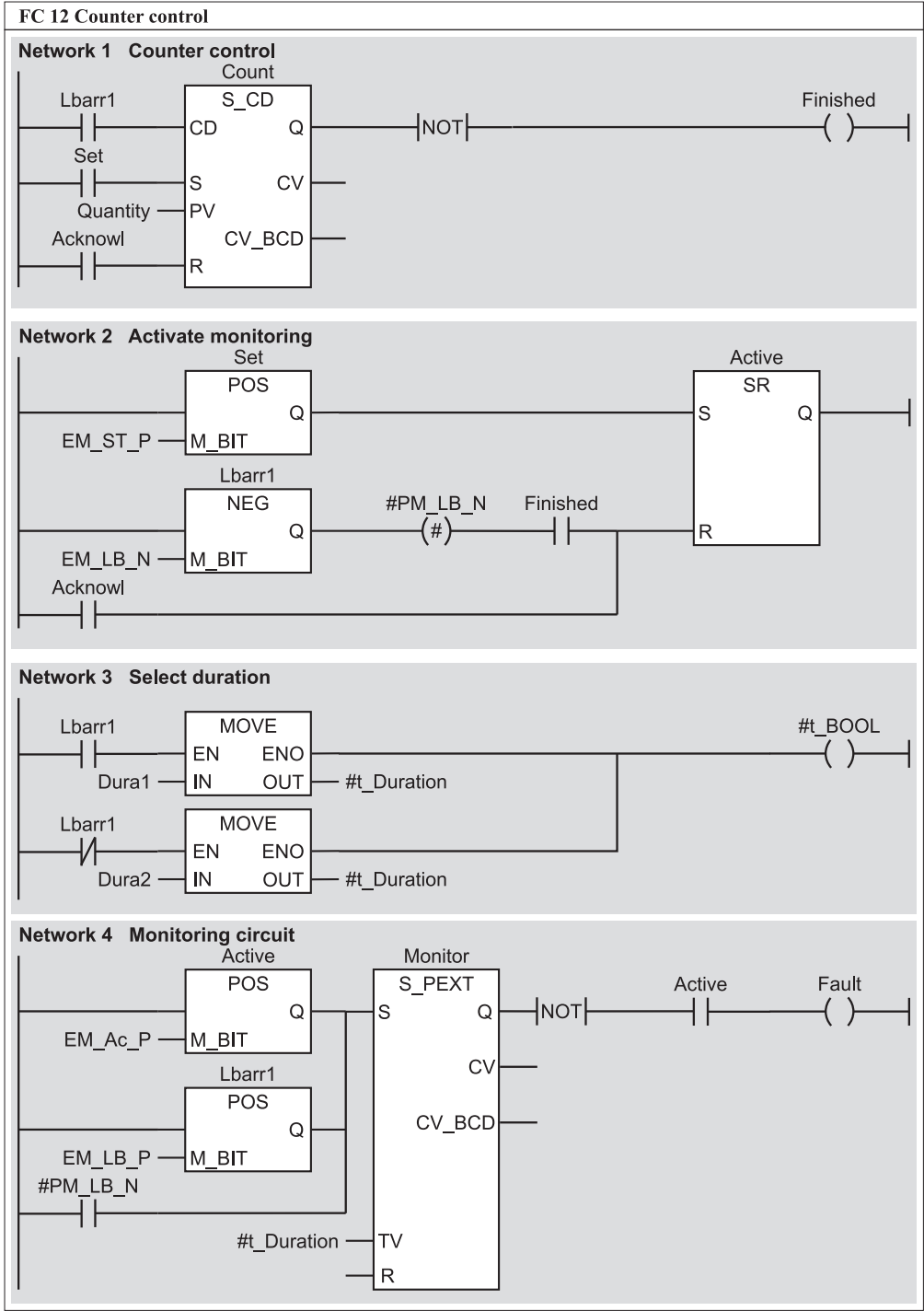
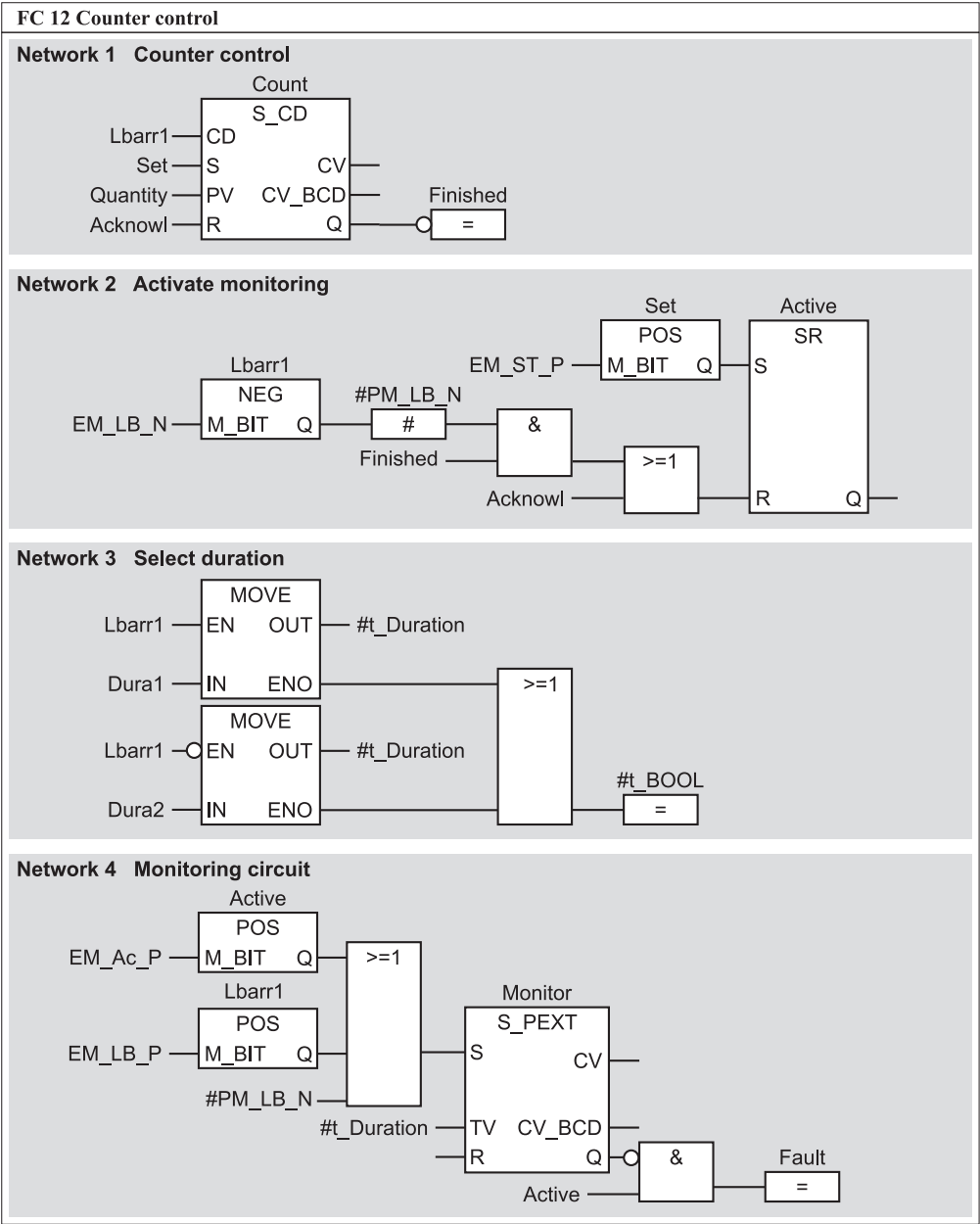


Figure 8.4 Programming Example for a Parts Counter (LAD)



Digital Functions

The digital functions process digital values predominantly with the data types INT, DINT and REAL, and thus extend the functionality of the PLC.

The **comparison functions** form a binary result from the comparison of two values. They take account of the data types INT, DINT and REAL.

You use the **arithmetic functions** to make calculations in your program. All the basic arithmetic functions in data types INT, DINT and REAL are available.

The **mathematical functions** extend the calculation possibilities beyond the basic arithmetic functions to include, for example, trigonometric functions.

Before and after performing calculations, you adapt the digital values to the desired data type using the **conversion functions**.

The **shift functions** allow justification of the contents of a variable by shifting to the right or left.

With **word logic**, you mask digital values by targeting individual bits and setting them to “1” or “0”.

The digital logic operations work mainly with values stored in data blocks. These can be global data blocks or instance data blocks if static local data are used. Chapter 18.2, “Block Functions for Data Blocks”, shows how to handle data blocks and gives the methods of addressing data operands.

09 Comparison Functions

Comparison for equal to, not equal to, greater than, greater than or equal to, less than, and less than or equal to

10 Arithmetic Functions

Basic arithmetic functions with data types INT, DINT and REAL

11 Mathematical Functions

Trigonometric functions; inverse trigonometric functions; squaring, square-root extraction, exponentiation, and logarithms

12 Conversion Functions

Conversion from INT/DINT to BCD and vice versa; conversion from DINT to REAL and vice versa with different forms of rounding; one's complement, negation, and absolute-value generation

13 Shift Functions

Shifting to left and right, by word and doubleword, shifting with correct sign; rotating to left and right

14 Word Logic

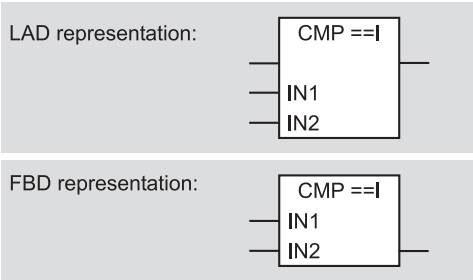
AND, OR, exclusive OR; word and doubleword combinations

9 Comparison Functions

The comparison functions compare two digital variables of data type INT, DINT or REAL for equal to, not equal to, greater than or greater than or equal to, less than, or less than or equal to. The comparison result is then available as binary value (Table 9.1).

9.1 Processing a Comparison Function

Comparison box
(in example: comparison for equal to INT)



Representation LAD

In addition to the (unlabeled) binary input, the box for a comparison function has two inputs, IN1 and IN2, and an (unlabeled) binary output. The “header” in the box identifies a comparison operation (CMP for compare) and the type of

comparison performed (CMP ==I, for instance, stands for the comparison of two INT numbers for equal to).

You can arrange a comparator in a rung in place of a contact. The unlabeled input and the unlabeled output establish the connection to the other (binary) program elements.

The values to be compared are at inputs IN1 and IN2 and the comparison result is the output. A successful comparison is equivalent to a closed contact (“power” flows through the comparator). If the comparison is not successful, the contact is open. The comparator’s output must always be interconnected.

Representation FBD

The box for a comparison has two inputs, IN1 and IN2, and an unlabeled binary output. The “header” in the box identifies the comparison performed (CMP ==I, for example, stands for the comparison of two INT numbers for equal to).

The values to be compared are at inputs IN1 and IN2 and the result of the comparison is at the output. If the comparison is successful, the comparator output shows signal state “1”; otherwise, it is “0”. It must always be interconnected.

Table 9.1 Overview of the Comparison Functions

Comparison Function	Comparison According to Data Type		
	INT	DINT	REAL
Comparison for equal to	CMP ==I	CMP ==D	CMP ==R
Comparison for not equal to	CMP <>I	CMP <>D	CMP <>R
Comparison for greater than	CMP >I	CMP >D	CMP >R
Comparison for greater than or equal to	CMP >=I	CMP >=D	CMP >=R
Comparison for less than	CMP <I	CMP <D	CMP <R
Comparison for less than or equal to	CMP <=I	CMP <=D	CMP <=R

Data types

The data type of the inputs in a comparison function depends on that function. For example, the inputs are of type REAL in the comparison function CMP >R (compare REAL numbers for greater than). Variables must be of the same data type as the inputs. When using operands with absolute addresses, the operand widths must accord with the data types. For example, you can use a word operand for data type INT.

You can find the bit assignments for the data formats in Chapter 3.5.4, “Elementary Data Types”.

A comparison between REAL numbers is not true if one or both REAL numbers are invalid. In addition, status bits OS and OV are set. You can find out how the comparison functions set the remaining status bits in Chapter 15, “Status Bits”.

Examples

Figure 9.1 provides an example for each of the data types. A comparison function carries out a comparison according to the characteristics specified even when no data types are declared when using operands with absolute addresses.

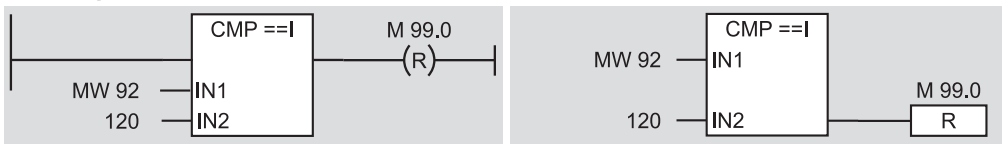
In the case of incremental programming, you will find the comparison functions in the Program Elements Catalog (with VIEW → OVERVIEWS [Ctrl - K] or INSERT → PROGRAM ELEMENTS) under “Comparator”.

Comparison function in a rung (LAD)

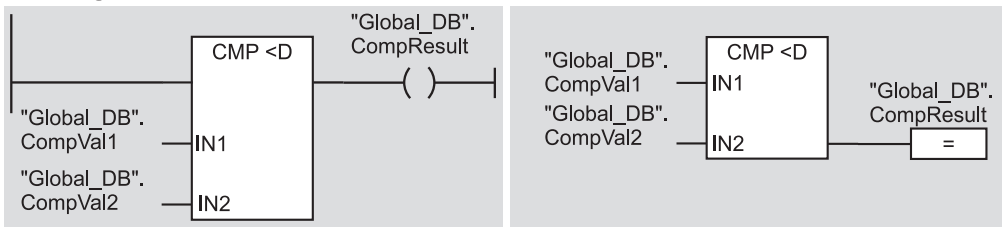
You can use the comparison function in a rung in place of a contact.

You can connect contacts before and after the comparison function in series and in parallel. The comparison boxes themselves can also be

Comparison according to INT Memory bit M 99.0 is reset if the value in memory word MW 92 is equals to 120; otherwise it is not.



Comparison according to DINT The variable “CompResult” in data block “Global_DB” is set if variable “CompVal1” is less than “CompVal2”, otherwise it is reset.



Comparison according to REAL If the variable #ActVal is greater than or equal to the variable #Calibra, #Recali is set; otherwise it is not.

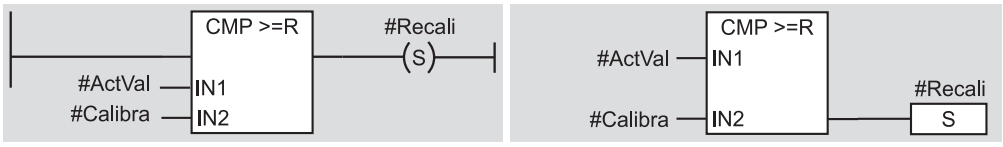


Figure 9.1 Comparison Function Examples

connected in series or in parallel. In the case of comparison functions connected in series, both comparisons must be successful for power to flow in the rung. In the case of comparators connected in parallel, only one compare condition need be fulfilled for power to flow in the parallel circuit.

You can find further examples of the representation and arrangement of comparison functions in the program “Digital Functions” (function block FB 109) in the library “LAD_Book” that you can download from the publisher's Website (see page 8).

Comparison function in a logic circuit (FBD)

You can position the comparison function at any binary input of a program element. The result of the comparison can be subsequently combined with binary functions.

You can find further examples of the representation and arrangement of comparison functions in the program “Digital Functions” (function block FB 109) in the library “FBD_Book” that you can download from the publisher's Website (see page 8).

9.2 Description of the Comparison Functions

Comparison for equal to

The “comparison for equal to” interprets the contents of the input variables in accordance with the data type specified in the comparison function and checks to see if the two values are equal. The compare condition is fulfilled (“power” flows through the comparator output or the RLO is “1”) when the two variables have the same value.

If, in the case of a REAL comparison, one or both input variables are invalid, the comparison is not successful. Status bits OV and OS are also set.

Comparison for not equal to

The “comparison for not equal to” interprets the contents of the input variables in accordance

with the data type specified in the comparison function and checks to see if the two values differ. The comparison is successful (“power” flows through the comparator output or the RLO is “1”) when the two variables have different values.

If, in the case of a REAL comparison, one or both input variables are invalid, the comparison is not successful. In addition, status bits OV and OS are set.

Comparison for greater than

The “comparison for greater than” interprets the contents of the input variables in accordance with the data type specified in the comparison function and checks to see if the value at input IN1 is greater than the value at IN2. If this is the case, the comparison is successful (“power” flows through the comparator output or the RLO is “1”).

If, in the case of a REAL comparison, one or both input variables are invalid, the comparison is not successful. In addition, status bits OV and OS are set.

Comparison for greater than or equal to

The “comparison for greater than or equal to” interprets the contents of the input variables in accordance with the data type specified in the comparison function and checks to see if the value at input IN1 is greater than or equal to the value at input IN2. If this is the case, the comparison is successful (“power” flows at the comparator output or the RLO is “1”).

If, in the case of a REAL comparison, one or both input variables are invalid, the comparison is not successful. In addition, status bits OV and OS are set.

Comparison for less than

The “comparison for less than” interprets the contents of the input variables in accordance with the data type specified in the comparison function and checks to see if the value at input IN1 is less than the value at input IN2. If this is the case, the comparison is successful (“power” flows at the comparator output or the RLO is “1”).

If, in the case of a REAL comparison, one or both input variables are invalid, the comparison is not successful. In addition, status bits OV and OS are set.

Comparison for less than or equal to

The “comparison for less than or equal to” interprets the contents of the input variables in accordance with the data type specified in the

comparison function and checks to see if the value at input IN1 is less than or equal to the value at input IN2. If this is the case, the comparison is successful (“power” flows at the comparator output or the RLO is “1”).

If, in the case of a REAL comparison, one or both input variables are invalid, the comparison is not successful. In addition, status bits OV and OS are set.

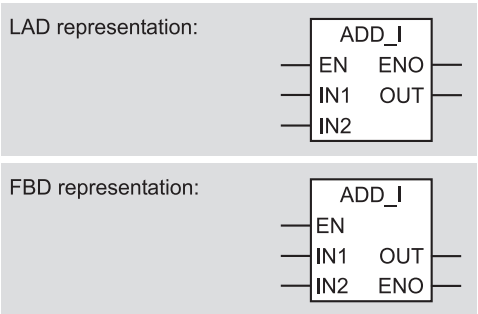
10 Arithmetic Functions

The arithmetic functions combine two values in accordance with the basic arithmetical operations of addition, subtraction, multiplication, and division. You can use the arithmetic functions on variables of type INT, DINT, and REAL (Table 10.1).

10.1 Processing an Arithmetic Function

Representation

Arithmetic box
(in example: addition with INT)



In addition to enable the input EN and the enable output ENO, a box for an arithmetic function has two inputs, IN1 and IN2, and an

output, OUT. The “header” in the box identifies the arithmetic function executed (ADD_I, for instance, stands for the addition of INT numbers).

The values to be combined are at inputs IN1 and IN2, and the result of the calculation is at output OUT. The inputs and the output have different data types, depending on the arithmetic function. For example, in the case of the arithmetic function ADD_R (addition of REAL numbers), the inputs and the output are of data type REAL. The variables applied must be of the same data type as the inputs or the output. If you use absolute addresses for the operands, the operand widths must be matched to the data types. For example, you can use a word operand for data type INT.

You can find a description of the individual bits in each data format in Chapter 3.5.4, “Elementary Data Types”.

Function

The arithmetic function is executed if “1” is present at the enable input (“power” flows in input EN). If an error occurs during the calculation, the enable output is set to “0”; otherwise, it is set to “1”. If execution of the function is not enabled (EN = “0”), the calculation does not take place, and ENO is also “0”.

Table 10.1 Overview of Arithmetic Functions

Arithmetic function	With data type		
	INT	DINT	REAL
Addition	ADD_I	ADD_DI	ADD_R
Subtraction	SUB_I	SUB_DI	SUB_R
Multiplication	MUL_I	MUL_DI	MUL_R
Division with quotient as result	DIV_I	DIV_DI	DIV_R
Division with remainder as result	-	MOD_DI	-

IF EN == "1" or not wired		ELSE
THEN		
OUT := IN1 Cfct IN2		
IF error occurred		
THEN	ELSE	
ENO := "0"	ENO := "1"	
		ENO := "0"

with Cfcf as calculation function

If the Master Control Relay (MCR) is activated, output OUT is set to zero when the arithmetic function is processed (EN = "1"). The MCR does not affect the ENO output.

The following errors can occur during execution of an arithmetic function:

- Range violation (overflow) in INT and DINT calculations
- Underflow and overflow in a REAL calculation

- Invalid REAL number in a REAL calculation

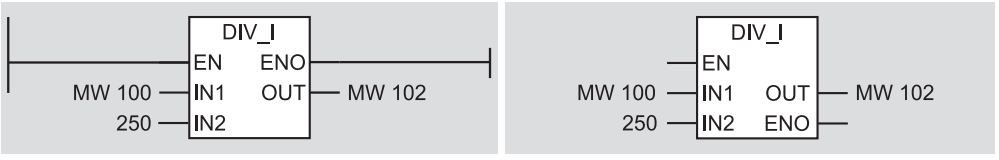
See Chapter 15, "Status Bits", to find out how the arithmetic functions set the various status bits.

Examples

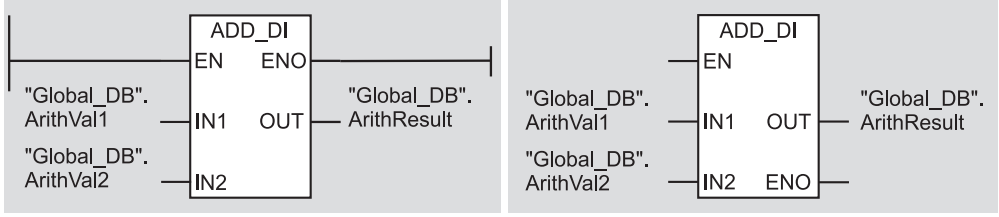
Figure 10.1 shows an example for each data type. An arithmetic function executes a calculation in accordance with the characteristic specified, even if no data types have been declared when using operands with absolute addresses.

In the case of incremental programming, you will find the arithmetic functions in the Program Element Catalog (with VIEW → OVERVIEWS [Ctrl - K] or INSERT → PROGRAM ELEMENTS) under "Integer function" (INT and DINT calculations) and under "Floating-Point fct." (REAL calculations).

Addition according to INT The value in memory word MW 100 is divided by 250; the integer result is stored in memory word MW 102.



Addition according to DINT The values in variables "ArithVal1" and "ArithVal2" are added and the result stored in variable "ArithResult". All variables are stored in the data block.



Addition according to REAL The variable #ActVal and \Factor are multiplied; the product is transferred to variable #Display.

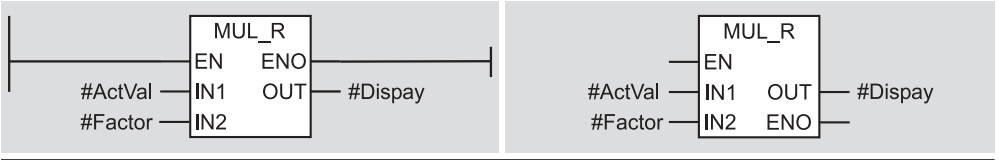


Figure 10.1 Examples of Arithmetic Functions

Arithmetic function in a rung (LAD)

You can connect contacts in series and in parallel before the EN input and after the ENO output.

The arithmetic box itself may be placed after a T-branch and in a branch that leads directly to the left power rail. This branch can also have contacts before the EN input and it need not be the uppermost branch.

Direct connection to the left power rail means that you can connect arithmetic boxes in parallel. When you connect boxes in parallel, you need a coil to terminate the rung. If you have not provided any error evaluation, assign a “dummy” operand to the coil, for example a temporary local data bit.

You can connect arithmetic boxes in series. If the ENO output of the preceding box leads to the EN input of the subsequent box, the subsequent box is processed only if the preceding box has been completed without errors. If you want to use the result from the preceding box as input value for the next box, variables from the temporary local data area make convenient intermediate buffers.

If you arrange several arithmetic boxes in one rung (parallel to the left power rail, then further in series), the boxes in the uppermost branch are processed from left to right, followed by the boxes in the second branch from left to right, and so on.

You can find further examples of the representation and arrangement of arithmetic functions in the program “Digital Functions” (FB 110) in the library “LAD_Book” that you can download from the publisher's Website (see page 8).

Arithmetic function in a logic circuit (FBD)

If you want to process the arithmetic box in dependence on specific conditions, you can arrange binary logic operations before the EN input. You can interconnect the ENO output with binary inputs of other functions; for example, you can arrange arithmetic boxes in series, whereby the ENO output of the preceding box leads to the EN input of the following box. If you want to use the calculation result from the preceding box as input value for a subsequent

box, variables from the temporary local data area make convenient intermediate buffers.

EN and ENO need not be wired.

You can find further examples of the representation and arrangement of arithmetic functions in the program “Digital Functions” (FB 110) in the library “FBD_Book” that you can download from the publisher's Website (see page 8).

10.2 Calculating with Data Type INT**INT addition**

The function ADD_I interprets the values at inputs IN1 and IN2 as numbers of data type INT. It adds the two numbers and stores the sum in output OUT.

After execution of the calculation, status bits CC0 and CC1 indicate whether the sum is negative, zero, or positive. Status bits OV and OS indicate any range violations.

INT subtraction

The function SUB_I interprets the values at inputs IN1 and IN2 as numbers of data type INT. It subtracts the value at IN2 from the value at IN1 and stores the difference in output OUT.

After execution of the calculation, status bits CC0 and CC1 indicate whether the difference is negative, zero, or positive. Status bits OV and OS indicate any range violations.

INT multiplication

The function MUL_I interprets the values at inputs IN1 and IN2 as numbers of data type INT. It multiplies the two numbers and stores the product in output OUT.

After execution of the calculation, status bits CC0 and CC1 indicate whether the product is negative, zero, or positive. Status bits OV and OS indicate any INT range violations.

INT division

The function DIV_I interprets the values at inputs IN1 and IN2 as numbers of data type INT. It divides the values at input IN1 (dividend) by the value at input IN2 (divisor) and supplies the quotient at output OUT. It is the

integer result of the division. The quotient is zero if the dividend is equal to zero and the divisor is not equal to zero or if the absolute value of the dividend is less than the absolute value of the divisor. The quotient is negative if the divisor is negative.

After execution of the calculation, status bits CC0 and CC1 indicate whether the quotient is negative, zero, or positive. Status bits OV and OS indicate any range violations. Division by zero produces zero as quotient and sets status bits CC0, CC1, OV and OS to “1”.

10.3 Calculating with Data Type DINT

DINT addition

The function `ADD_DI` interprets the values at inputs IN1 and IN2 as numbers of data type DINT. It adds the two numbers and stores the sum in output OUT.

After execution of the calculation, status bits CC0 and CC1 indicate whether the sum is negative, zero, or positive. Status bits OV and OS indicate any range violations.

DINT subtraction

The function `SUB_DI` interprets the values at inputs IN1 and IN2 as numbers of data type DINT. It subtracts the value at input IN2 from the value at input IN1 and stores the difference in output OUT.

After execution of the calculation, status bits CC0 and CC1 indicate whether the difference is negative, zero, or positive. Status bits OV and OS indicate any range violations.

DINT multiplication

The function `MUL_DI` interprets the values at inputs IN1 and IN2 as numbers of data type DINT. It multiplies the two numbers and stores the product in output OUT.

After execution of the calculation, status bits CC0 and CC1 indicate whether the product is negative, zero, or positive. Status bits OV and OS indicate any range violations.

DINT division with quotient as result

The function `DIV_DI` interprets the values at inputs IN1 and IN2 as numbers of data type DINT. It divides the value at input IN1 (dividend) by the value at input IN2 (divisor) and stores the quotient in output OUT. It is the integer result of the division. The quotient is zero if the dividend is equal to zero and the divisor is not equal to zero or if the absolute value of the dividend is less than the absolute value of the divisor. The quotient is negative if the divisor is negative.

After execution of the calculation, status bits CC0 and CC1 indicate whether the quotient is negative, zero, or positive. Status bits OV and OS indicate any range violations. Division by zero produces zero as quotient and sets status bits CC0, CC1, OV and OS to “1”.

DINT division with remainder as result

The function `MOD_DI` interprets the values at inputs IN1 and IN2 as numbers of data type DINT. It divides the value at input IN1 (dividend) by the value at input IN2 (divisor) and stores the remainder of the division in output OUT. The remainder is what is left over from the division; it does not correspond to the decimal places. If the dividend is negative, the remainder is also negative.

After execution of the calculation, status bits CC0 and CC1 indicate whether the remainder is negative, zero, or positive. Status bits OV and OS indicate any range violations. Division by zero produces zero as remainder and sets status bits CC0, CC1, OV and OS to “1”.

10.4 Calculating with Data Type REAL

REAL numbers are represented internally as floating-point numbers with two number ranges: One range with full accuracy (“normalized” floating-point numbers) and one range with limited accuracy (“denormalized” floating-point numbers; also see Chapter 3.5.4, “Elementary Data Types”). S7-400 CPUs calculate in both ranges, S7-300 CPUs only in the full-accuracy range. If an S7-300 CPU carries out a calculation whose result is in the limited-accu-

racy range, zero is returned as result and a range violation reported.

REAL addition

The function `ADD_R` interprets the values at inputs `IN1` and `IN2` as numbers of data type `REAL`. It adds the two numbers and stores the sum in output `OUT`.

After execution of the calculation, status bits `CC0` and `CC1` indicate whether the sum is negative, zero, or positive. Status bits `OV` and `OS` indicate any range violations.

In the case of an illegal calculation (one of the input values is an invalid `REAL` number or you attempt to add $+\infty$ and $-\infty$), `ADD_R` returns an invalid value at output `OUT` and sets status bits `CC0`, `CC1`, `OV` and `OS` to “1”.

REAL subtraction

The function `SUB_R` interprets the values at inputs `IN1` and `IN2` as numbers of data type `REAL`. It subtracts the number at input `IN2` from the number at input `IN1` and stores the difference in output `OUT`.

After execution of the calculation, status bits `CC0` and `CC1` indicate whether the difference is negative, zero, or positive. Status bits `OV` and `OS` indicate any range violations.

In the case of an illegal calculation (one of the input values is an invalid `REAL` number or you attempt to subtract $+\infty$ from $+\infty$), `SUB_R` returns an invalid value at output `OUT` and sets status bits `CC0`, `CC1`, `OV` and `OS` to “1”.

REAL multiplication

The function `MUL_R` interprets the values at inputs `IN1` and `IN2` as numbers of data type `REAL`. It multiplies the two numbers and stores the product in output `OUT`.

After execution of the calculation, status bits `CC0` and `CC1` indicate whether the product is negative, zero, or positive. Status bits `OV` and `OS` indicate any range violations.

In the case of an illegal calculation (one of the input values is an invalid `REAL` number or you attempt to multiply ∞ and 0), `MUL_R` returns an invalid value at output `OUT` and sets status bits `CC0`, `CC1`, `OV` and `OS` to “1”.

REAL division

The function `DIV_R` interprets the values at inputs `IN1` and `IN2` as numbers of data type `REAL`. It divides the number at input `IN1` (dividend) by the number at input `IN2` (divisor) and stores the quotient in output `OUT`.

After execution of the calculation, status bits `CC0` and `CC1` indicate whether the quotient is negative, zero, or positive. Status bits `CC0` and `CC1` indicate any range violations.

In the case of an illegal calculation (one of the input values is an invalid `REAL` number or you attempt to divide ∞ by ∞ or 0 by 0), `DIV_R` returns an invalid value at output `OUT` and sets status bits `CC0`, `CC1`, `OV` and `OS` to “1”.

11 Mathematical Functions

The following mathematical functions are available in LAD and FBD:

- ▷ Sine, cosine, tangent
- ▷ Arc sine, arc cosine, arc tangent
- ▷ Squaring, square-root extraction
- ▷ Exponential function to base e, natural logarithm

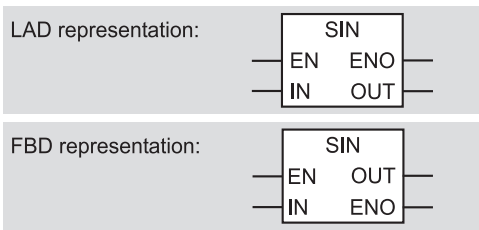
All mathematical functions process REAL numbers.

11.1 Processing a Mathematical Function

Representation

The box for a mathematical function has an input IN and an output OUT in addition to the enable input EN and the enable output ENO. The “header” in the box identifies the mathematical function executed (for example, SIN stands for sine).

Math box
(in example: sine)



The input value is at input IN and the result of the mathematical function is at output OUT. Input and output are of data type REAL. Operands referenced with absolute addresses must be doubleword operands.

See Chapter 3.5.4, “Elementary Data Types”, for a description of the bits in REAL format.

Function

The mathematical function is executed if “1” is present at the enable input or if “power” flows in input EN. If an error occurs in the calculation, the enable input is set to “0”; otherwise it is set to “1”. If execution of the function is not enabled (EN = “0”), the calculation does not take place and ENO is also “0”.

IF EN == “1” or not wired			ELSE
THEN			
OUT := Mfct (IN)			
IF error occurred			
THEN	ELSE		
ENO := “0”	ENO := “1”	ENO := “0”	

with Mfct as mathematical function

If the Master Control Relay (MCR) is active, output OUT is set to zero when the mathematical function is processed (EN = “1”). The MCR does not affect the ENO.

The following errors can occur in a mathematical function:

- ▷ Range violation (underflow and overflow)
- ▷ No valid REAL number as input value

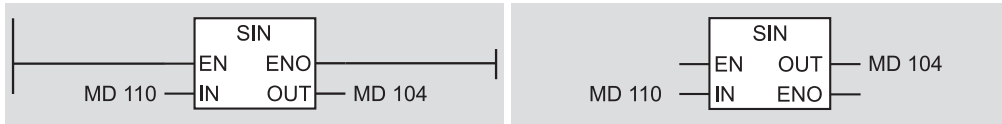
Chapter 15, “Status Bits”, explains how the mathematical functions set the status bits.

Examples

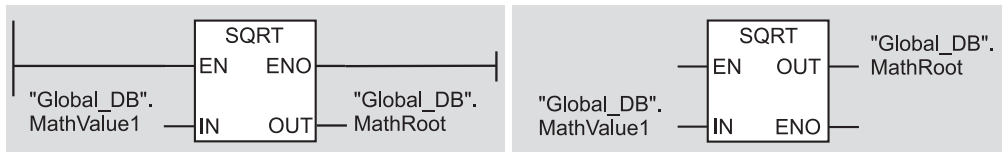
Figure 11.1 shows three examples of mathematical functions. A mathematical function performs the calculation in accordance with REAL even if no data types have been declared when using operands with absolute addresses.

In the case of incremental programming, you will find the mathematical functions in the Program Elements Catalog (with VIEW → OVERVIEWS [Ctrl - K] or INSERT → PROGRAM ELEMENTS) under “Floating-Point fct.”.

Sine The value in memory doubleword MD 110 contains an angle in radian measure. The sine of this angle is generated and stored in memory doubleword MD 104.



Square root The square root of the value in variable “MathValue1” is generated and stored in the variable “MathRoot”.



Exponent The variable #Result contains the power of e and #Exponent.



Figure 11.1 Examples of Mathematical Functions

Mathematical function in a rung (LAD)

You can connect contacts in series and in parallel before input EN and after output ENO.

The mathematics box itself may be placed after a T-branch or in a branch that leads directly to the left power rail. This branch can also have contacts before input EN and need not be the uppermost branch.

The direct connection to the left power rail allows you to connect mathematics boxes in parallel. When connecting boxes in parallel, you require a coil to terminate the rung. If you have not provided error evaluation, assign a “dummy” operand to the coil, for example a temporary local data bit.

You can connect mathematics boxes in series. If the ENO output of the preceding box leads to the EN input of the subsequent box, the subsequent box is processed only if the preceding box has been completed without errors. If you want to use the result from the preceding box as the input value for a subsequent box, variables from the temporary local data area make convenient intermediate buffers.

If you arrange several mathematics boxes in one rung (parallel to the left power rail and then

further in series), the boxes in the uppermost branch are the first to be processed from left to right, followed by the boxes in the second branch from left to right, and so on.

You can find further examples of the representation and arrangement of mathematical functions in the program “Digital Functions” (FB 111) in the library “LAD_Book” that you can download from the publisher’s Website (see page 8).

Mathematical function in a logic circuit (FBD)

If you want to have a mathematics box processed in dependence on specific conditions, you can program binary logic operations before the EN input. You can connect the ENO output with binary inputs from other functions. For example, you can arrange the mathematics boxes in series, whereby the ENO output of the preceding box leads to the EN input of the following box. If you want to use the calculation result of the preceding box as input value to a subsequent box, variables from the temporary local data area make good intermediate buffers.

EN and ENO need not be wired.

You can find further examples of the representation and arrangement of mathematical functions in the program “Digital Functions” (FB 111) in the library “FBD_Book” that you can download from the publisher’s Website (see page 8).

11.2 Trigonometric Functions

The trigonometric functions

SIN	Sine,
COS	Cosine and
TAN	Tangent

assume an angle in radian measure as a REAL number at the input.

Two units are conventionally used for giving the size of an angle: degrees from 0° to 360° and radian measure from 0 to 2π (where π = +3.141593e+00). Both can be converted proportionally. For example, the radian measure for a 90° angle is π/2, or +1.570796e+00. With values greater than 2π (+6.283185e+00), 2π or a multiple of 2π is subtracted until the input value for the trigonometric function is less than 2π.

Example (Figure 11.2 or Figure 11.3, Network 4): Calculating the idle power $P_s = U \cdot I \cdot \sin(\varphi)$

Table 11.1 Range of arc functions

Function	Permissible Range	Value Returned
ASIN	−1 to +1	−π/2 to +π/2
ACOS	−1 to +1	0 to π
ATAN	Full range	−π/2 to +π/2

11.3 Arc Functions

The arc functions (inverse trigonometric functions)

ASIN	Arc sine,
ACOS	Arc cosine and
ATAN	Arc tangent

are the inverse functions of the corresponding trigonometric functions. They assume a REAL number in a specific range at input IN and return an angle in the radian measure (Table 11.1).

If the permissible value range is exceeded at input IN, the arc function returns an invalid REAL number and ENO = “0” and sets status bits CC0, CC1, OV and OS to “1”.

11.4 Miscellaneous Mathematical Functions

The following mathematical functions are also available

SQR	Compute the square of a number,
SQRT	Compute the square root of a number,
EXP	Compute the exponent to base e and
LN	Find the natural logarithm (logarithm to base e).

Computing the square

The SQR function squares the value at input IN and stores the result in output OUT.

Example: See “Computing the square root”.

Computing the square root

The SQRT function extracts the square root of the value at input IN and stores the result in output OUT. If the value at input IN is less than zero, SQRT sets status bits CC0, CC1, OV and OS to “1” and returns an invalid REAL number. If the value at input IN is −0 (minus zero), −0 is returned.

Example: $c = \sqrt{a^2 + b^2}$

Figure 11.2 or Figure 11.3, Network 5: First, the squares of variables a and b are found and then added. Finally, the square root is extracted from the sum. Temporary local data are used as intermediate memory.

(If you have declared a or b as a local variable, you must precede it with # so that the Editor recognizes it as a local variable; if a or b is a global variable, it must be placed in quotation marks.)

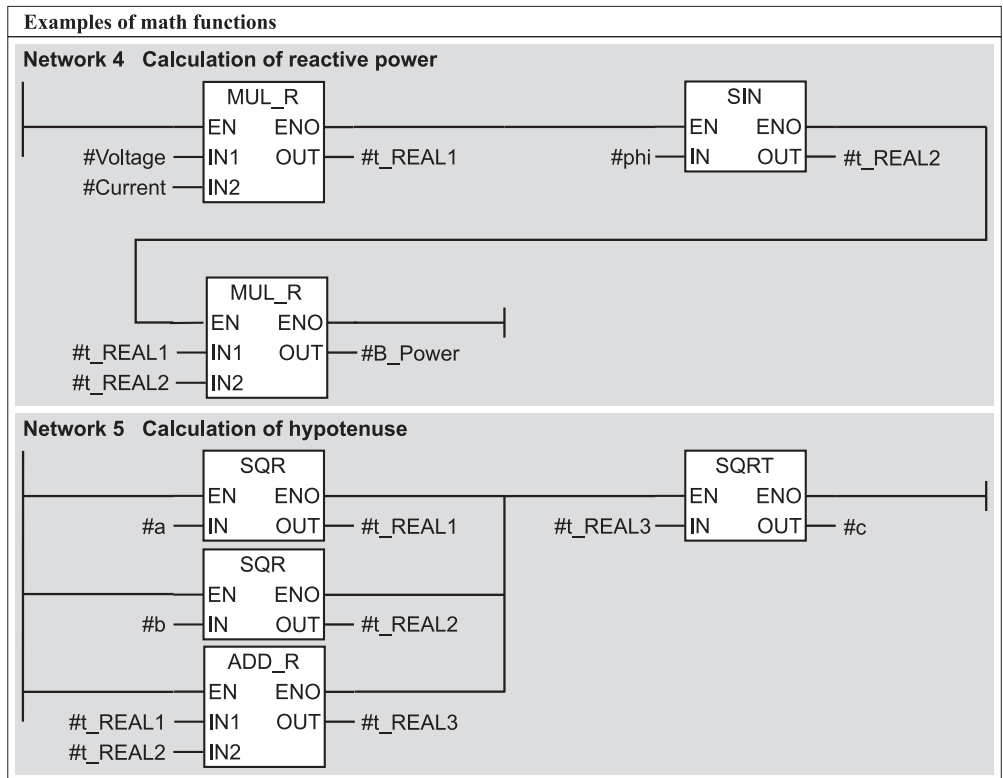


Figure 11.2 Examples of Mathematical Functions (LAD)

Computing the exponent to base e

The EXP function computes the exponential value to base e ($= 2.718282e+00$) and the value at input IN (e^{IN}) and stores the result in output OUT.

You can calculate any exponential value using the formula

$$a^b = e^{b \ln a}$$

Finding the natural logarithm

The LN function finds the natural logarithm to base e ($= 2.718282e+00$) from the number at input N and stores it in output OUT. If the value at input IN is less than or equal to zero, LN sets status bits CC0, CC1, OV and OS to “1” and returns an invalid REAL number.

The natural logarithm is the inverse function of the exponential function: If $y = e^x$ then $x = \ln y$.

To find any logarithm, use the formula

$$\log_b a = \frac{\log_n a}{\log_n b}$$

where b or n is any base. If you make $n = e$, you can find a logarithm to any base using the natural logarithm:

$$\log_b a = \frac{\ln a}{\ln b}$$

In the special case for base 10, the formula is as follows:

$$\lg a = \frac{\ln a}{\ln 10} = 0.4342945 \cdot \ln a$$

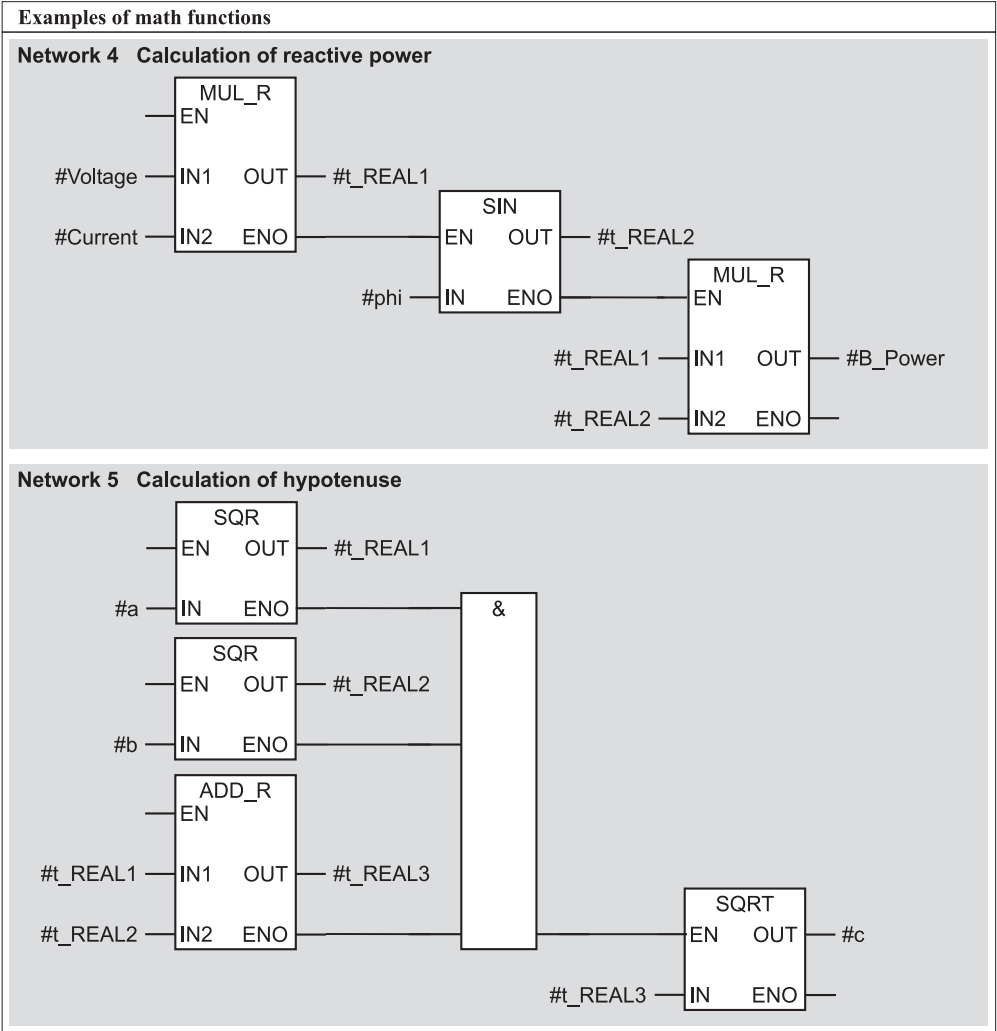


Figure 11.3 Examples of Mathematical Functions (FBD)

12 Conversion Functions

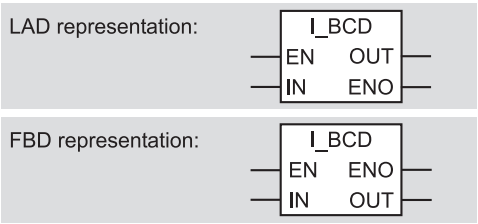
The conversion functions convert the data type of a variable. Figure 12.1 provides an overview of the data type conversions described in this chapter.

12.1 Processing a Conversion Function

Representation

In addition to the enable input EN and the enable output ENO, the box for a conversion function has an input IN and an output OUT. The “header” in the box identifies the conversion function executed (for example, I_BCD stands for the conversion of INT to BCD).

Conversion box
(in example: INT to BCD)



The value to be converted is at input IN and the result of the conversion is at output OUT. The data type of the input and that of the output depend on the conversion function. In the conversion function DI_R (DINT to REAL), for instance, the input is of type DINT and the output of type REAL. The variables applied must be of the same data type as the input or the output. If you use operands with absolute addresses, the sizes of the operands must be matched to the data types; for example, you can use a word operand for data type INT.

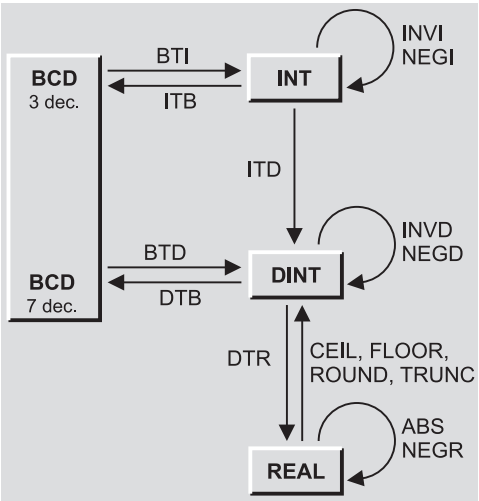


Figure 12.1 Overview of the Conversion Functions

You can find the bit descriptions for the data formats in Chapter 3.5.4, “Elementary Data Types”.

Function

The conversion function is executed if “1” is present at the enable input (if current flows in input EN). If an error occurs during conversion, the enable output ENO is set to “0”; otherwise, it is set to “1”. If execution of the function is not enabled (EN = “0”), the conversion does not take place and ENO is also “0”.

IF EN == “1” or not wired		
THEN		ELSE
OUT := Confct (IN)		
IF error occurred		
THEN	ELSE	
ENO := “0”	ENO := “1”	ENO := “0”

with Confct as conversion function

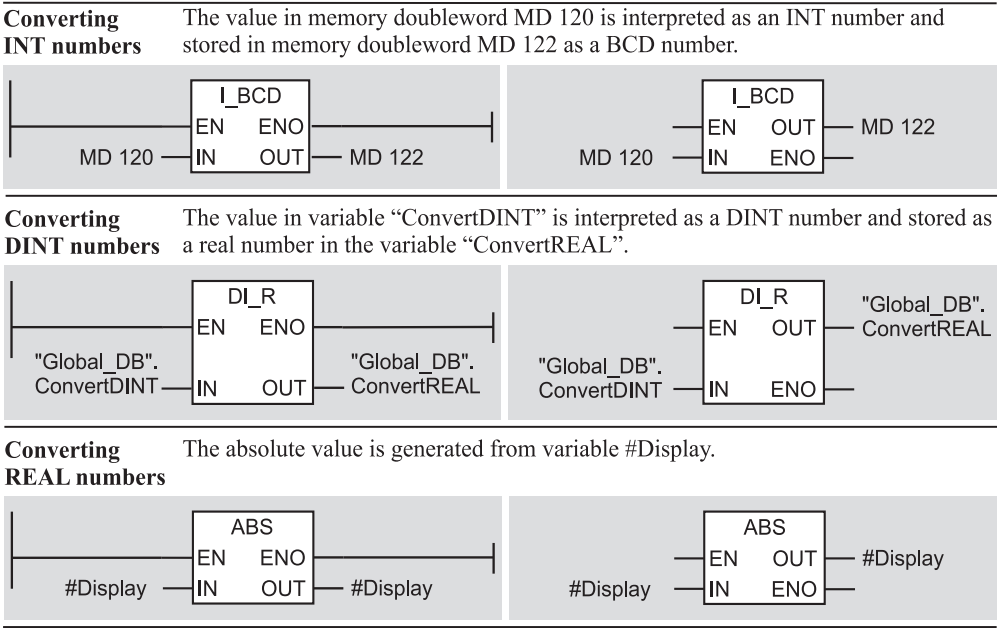


Figure 12.2 Examples of Conversion Functions

If the Master Control Relay (MCR) is active, output OUT is set to zero when the conversion function is processed (EN = "1"). The MCR has no effect on output ENO.

Not all conversion functions report errors. An error occurs only if the permissible number range is exceeded (I_BCD, DI_BCD) or an invalid REAL number is specified (FLOOR, CEIL, ROUND, TRUNC).

If the input value for a BCD_I or BCD_DI conversion contains a pseudo tetrad, program execution is interrupted and error organization block OB 121 (programming errors) called.

Chapter 15, "Status Bits", explains how the conversion functions set the status bits.

Figure 12.2 shows one example for each data type. A conversion function converts according to the specific characteristic even if no data types have been declared when using operands with absolute addresses.

In the case of incremental programming, you will find the conversion functions in the Program Elements Catalog (with VIEW → OVER-

VIEWS [Ctrl - K] or INSERT → PROGRAM ELEMENTS) under "Converter".

Conversion function in a rung (LAD)

You can connect contacts in series and in parallel before the EN input and after the ENO output.

The conversion box itself may be placed after a T-branch or in a branch that leads directly to the left power rail. This branch can also have contacts before input EN and it need not be the uppermost branch.

With the direct connection to the left power rail you can thus connect conversion boxes in parallel. When connecting boxes in parallel, you require a coil to terminate the rung. If you have not provided error evaluation, assign a "dummy" operand to the coil, for instance a temporary local data bit.

You can connect conversion boxes in series. If the ENO output of the preceding box leads to the EN output of the subsequent box, the subsequent box is processed only if the preceding box has been completed without errors. If you

want to use the result from the preceding box as the input value for a subsequent box, variables from the temporary local data area make convenient intermediate buffers.

If you arrange several conversion boxes in one rung (parallel to the left power rail and then further in series), the boxes in the uppermost branch are processed first from left to right, followed by the boxes in the second branch from left to right, and so on.

You can find further examples of the representation and arrangement of conversion functions in the program “Digital Functions” (FB 111) in the library “LAD_Book” that you can download from the publisher's Website (see page 8).

Conversion function in a logic circuit (FBD)

If you want to process the conversion box in dependence on specific conditions, you can arrange binary logic operations before the EN input. You can connect the ENO output with binary inputs of other functions; for example, you can arrange conversion boxes in series, whereby the ENO output of the preceding box leads to the EN input of the subsequent box. If you want to use the result from the preceding box as input value for a subsequent box, variables from the temporary local data area make good temporary buffers.

EN and ENO need not be wired.

You can find further examples of the representation and arrangement of conversion functions in the program “Digital Functions” (FB 111) in the library “FBD_Book” that you can download from the publisher's Website (see page 8).

12.2 Conversion of INT and DINT Numbers

Table 12.1 shows the conversion functions for INT and DINT numbers. Variables of the specified data type or absolute-addressed operands of the relevant size must be applied to the inputs and outputs of the boxes (for example a word operand for data type INT).

Conversion from INT to DINT

The function `I_DI` interprets the value at input IN as a number of data type INT and transfers it to the low-order word of output OUT. The signal state of bit 15 (the sign) of the input is transferred to bits 16 to 31 of the high-order word of output OUT.

The conversion from INT to DINT reports no errors.

Conversion from INT to BCD

The function `I_BCD` interprets the value at input IN as a number of data type INT and converts it to a 3-decade BCD number at output OUT. The three right-justified decades represent the absolute value of the decimal number. The sign is in bits 12 to 15. If all bits are “0”, the sign is positive; if all bits are “1”, the sign is negative.

If the INT number is too large to be converted into BCD (> 999), the `I_BCD` function sets status bits OV and OS. The conversion does not take place.

Conversion from DINT to BCD

The function `DI_BCD` function interprets the value at input IN as a number of data type DINT and converts it to a 7-decade BCD number at output OUT. The seven right-justified decades represent the absolute value of the decimal number. The sign is in bits 28 to 31. If all bits are “0”, the sign is positive; if all bits are “1”, the sign is negative.

If the DINT number is too large to be converted to a BCD number (> 9 999 999), status bits OV and OS are set. The conversion does not take place.

Table 12.1 Conversion of INT and DINT Numbers

Data Type Conversion	Box	Data Type for Parameter	
		IN	OUT
INT to DINT	<code>I_DI</code>	INT	DINT
INT to BCD	<code>I_BCD</code>	INT	WORD
DINT to BCD	<code>DI_BCD</code>	DINT	DWORD
DINT to REAL	<code>DI_R</code>	DINT	REAL

Conversion from DINT to REAL

The function DI_R interprets the value at input IN as a number of data type DINT and converts it to a REAL number at output OUT.

Since a number in DINT format has a higher accuracy than a number in REAL format, rounding make take place during the conversion. The REAL number is then rounded to the next integer (in accordance with the ROUND function).

The DI_R function does not report errors.

12.3 Conversion of BCD Numbers

Table 12.2 shows the functions for converting BCD numbers. Variables of the specified type of absolute-addressed operands of the relevant size must be applied to the input and outputs of the boxes (for example a word operand for data type INT).

Table 12.2 Conversion of BCD Numbers

Data Type Conversion	Box	Data Type for Parameter	
		IN	OUT
BCD to INT	BCD_I	WORD	INT
BCD to DINT	BCD_DI	DWORD	DINT

Conversion from BCD to INT

The function BCD_I interprets the value at input IN as a 3-decade BCD number and converts it to an INT number at output OUT. The three right-justified decades represent the absolute value of the decimal number. The sign is in bits 12 to 15. If these bits are “0”, the sign is positive; if they are “1”, the sign is negative. Only bit 15 is taken into account in the conversion.

If the BCD number contains a pseudo tetrad (numerical value 10 to 15 or A to F in hexadecimal), the CPU signals a programming error and calls organization block OB 121 (synchronization error). If this block is not available, the CPU goes to STOP.

The function BCD_I sets no status bits.

Conversion from BCD to DINT

The function BCD_DI interprets the value at input IN as a 7-decade BCD number and converts it to an INT number at output OUT. The seven right-justified decades represent the absolute value of the decimal number. The sign is in bits 28 to 31. If these bits are “0”, the sign is positive; if they are “1”, the sign is negative. Only bit 31 is taken into account in the conversion.

If the BCD number contains a pseudo tetrad (numerical value 10 to 15 or A to F in hexadecimal), the CPU signals a programming error and calls organization block OB 121 (synchronization error). If this block is not available, the CPU goes to STOP.

The function BCD_I sets no status bits.

12.4 Conversion of REAL Numbers

There are several functions for converting a number in REAL format to DINT format (conversion of a fractional value to an integer) (Table 12.3). They differ as regards rounding. Variables of the specified data type or absolute-addressed doubleword operands must be applied to the inputs and outputs of the boxes.

Table 12.3 Conversion of REAL Numbers to DINT Numbers

Data Type Conversion with Rounding	Box	Data Type for Parameter	
		IN	OUT
To next higher integer	CEIL	REAL	DINT
To next lower integer	FLOOR	REAL	DINT
To next integer	ROUND	REAL	DINT
Without rounding	TRUNC	REAL	DINT

Rounding to the next higher integer

The function CEIL interprets the value at input IN as a number in REAL format and converts it to a number in DINT format at output OUT. CEIL returns an integer that is greater than or equal to the number to be converted.

Table 12.4 Rounding Modes for the Conversion of REAL Numbers

Input value		Result			
REAL	DW#16#	ROUND	CEIL	FLOOR	TRUNC
1.0000001	3F80 0001	1	2	1	1
1.00000000	3F80 0000	1	1	1	1
0.99999995	3F7F FFFF	1	1	0	0
0.50000005	3F00 0001	1	1	0	0
0.50000000	3F00 0000	0	1	0	0
0.49999996	3EFF FFFF	0	1	0	0
5.877476E-39	0080 0000	0	1	0	0
0.0	0000 0000	0	0	0	0
-5.877476E-39	8080 0000	0	0	-1	0
-0.49999996	BEFF FFFF	0	0	-1	0
-0.50000000	BF00 0000	0	0	-1	0
-0.50000005	BF00 0001	-1	0	-1	0
-0.99999995	BF7F FFFF	-1	0	-1	0
-1.00000000	BF80 0000	-1	-1	-1	-1
-1.0000001	BF80 0001	-1	-1	-2	-1

If the value at input IN exceeds or falls short of the range permissible for a number in DINT format or if it does not correspond to a number in REAL format, CEIL sets status bits OV and OS. Conversion does not take place.

If the value at input IN exceeds or falls short of the range permissible for a number in DINT format or if it does not correspond to a number in REAL format, ROUND sets status bits OV and OS. Conversion does not take place.

Rounding to the next lower integer

The function FLOOR interprets the value at input IN as a number in REAL format and converts it to a number in DINT format at output OUT. FLOOR returns an integer that is less than or equal to the number to be converted.

If the value at input IN exceeds or falls short of the range permissible for a number in DINT format or if it does not correspond to a number in REAL format, FLOOR sets status bits OV and OS. Conversion does not take place.

Rounding to the next integer

The function ROUND interprets the value at input IN as a number in REAL format and converts it to a number in DINT format at output OUT. ROUND returns the next integer. If the result lies exactly between an odd and an even number, the even number is given preference.

No rounding

The function TRUNC interprets the value at input IN as a number in REAL format and converts it to a number in DINT format at output OUT. TRUNC returns the integer component of the number to be converted; the fractional component is “truncated”.

If the value at input IN exceeds or falls short of the range permissible for a number in DINT format or if it does not correspond to a number in REAL format, TRUNC sets status bits OV and OS. Conversion does not take place.

Summary of conversions from REAL to DINT

Table 12.4 shows the different effects of the functions for converting from REAL to DINT. The range -1 to +1 has been chosen as example.

12.5 Miscellaneous Conversion Functions

Other available conversion functions are one's complement generation, negation, and absolute-value generation of a REAL number (Table 12.5). Variables of the specified data type or absolute-addressed operands of the relevant size must be applied to the inputs and outputs of the boxes (for example a doubleword operand for data type DINT).

One's complement INT

The function INV_I negates the value at input IN bit for bit and writes it to output OUT. INV_I replaces the zeroes with ones and vice versa. The function INV_I does not signal errors.

One's complement DINT

The function INV_DI negates the value at input IN bit for bit and writes it to output OUT. INV_DI replaces the zeroes with ones and vice versa. The function INV_DI does not signal errors.

Negation INT

The function NEG_I interprets the value at input IN as an INT number, changes the sign through two's complement generation, and writes the converted value to output OUT. NEG_I is identical to multiplication with -1 . The function NEG_I sets status bits CC0, CC1, OV and OS.

Negation DINT

The function NEG_DI interprets the value at input IN as a DINT number, changes the sign through two's complement generation, and writes the converted value to output OUT.

Table 12.5 Miscellaneous Conversion Functions

Conversion	Box	Data Type for Parameter	
		IN	OUT
One's complement INT	INV_I	INT	INT
One's complement DINT	INV_DI	DINT	DINT
Negation of an INT number	NEG_I	INT	INT
Negation of a DINT number	NEG_DI	DINT	DINT
Negation of a REAL number	NEG_R	REAL	REAL
Absolute value of a REAL number	ABS	REAL	REAL

NEG_DI is identical to multiplication by -1 . The function NEG_DI sets status bits CC0, CC1, OV and OS.

Negation REAL

The function NEG_R interprets the value at input IN as a REAL number, multiplies this number by -1 , and writes it to output OUT. NEG_R changes the sign of the mantissa even on an invalid REAL number. NEG_R does not signal errors.

Absolute-value generation REAL

The ABS function interprets the value at input IN as a REAL number, generates the absolute value from this number, and writes it to output OUT. ABS sets the sign of the mantissa to "0" even on an invalid REAL number. ABS does not signal errors.

13 Shift Functions

The shift functions shift the contents of a variable bit by bit to the left or right. The bits shifted out are either lost or are used to pad the other side of the variable. Table 13.1 provides an overview of the shift functions.

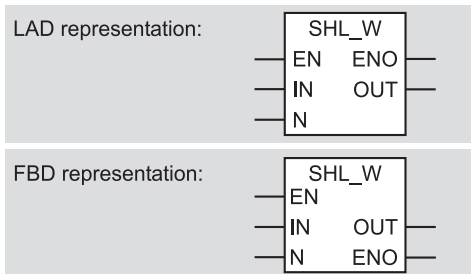
13.1 Processing a Shift Function

Representation

In addition to an enable input EN and an enable output ENO, the box for a shift function has an input IN, an input N, and an output OUT. The “header” in the box identifies the shift function executed (for example, SHL_W stands for shifting a word variable to the left).

Shift box

(in example: shift to left word by word)



The value to be shifted is at input IN, the number of places to be shifted is at input N, and the result is at output OUT. The input and the out-

put have different data types depending on the shift function. For example, input and output are of type DWORD for the shift function SHR_DW (shift a doubleword variable to the right). The variables applied must be of the same data type as the input or output. If you use operands with absolute addresses, the operand sizes must accord with the data types; for instance, you can use a word operand for data type INT. Input N has data type WORD for every shift function.

See Chapter 3.5.4, “Elementary Data Types”, for a description of the bits in each data format.

Function

The shift function is executed if “1” is present at the enable input or when “power” flows through input EN. ENO is then “1”. If execution of the function is not enabled (EN = “0”), the shift does not take place and ENO is also “0”.

IF EN == “1” or not wired	
THEN	ELSE
OUT := Sfct (IN, N)	
ENO := “1”	ENO := “0”

with Sfct as shift function

When the Master Control Relay (MCR) is activated, output OUT is set to zero when the shift function is processed (EN = “1”). The MCR does not affect output ENO.

Chapter 15, “Status Bits”, explains how the shift functions set the status bits.

Examples

Figure 13.1 gives one example each for various shift functions.

In the case of incremental programming, you will find the shift functions in the Program Elements Catalog (with VIEW → OVERVIEWS [Ctrl

Table 13.1 Overview of Shift Functions

Shift Functions	Word Variable	Doubleword Variable
Shift left	SHL_W	SHL_DW
Shift right	SHR_W	SHR_DW
Shift with sign	SHR_I	SHR_DI
Rotate left	-	ROL_DW
Rotate right	-	ROR_DW

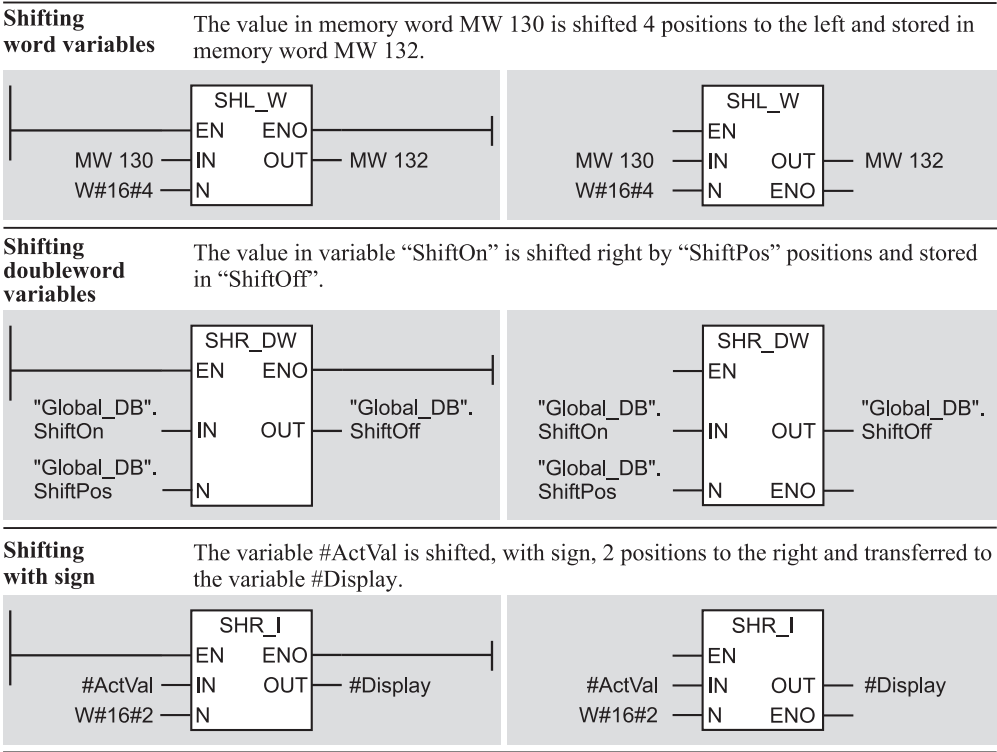


Figure 13.1 Examples of Shift Functions

- K] or INSERT → PROGRAM ELEMENTS) under “Shift/Rotate”.

Shift function in a rung (LAD)

You can connect contacts in series and in parallel before input EN and after output ENO.

The shift box itself may be placed after a T-branch or in a branch that leads directly to the left power rail. This branch can also have contacts before input EN and it need not be the uppermost branch.

The direct connection to the left power rail allows you to connect shift boxes in parallel. When connecting boxes in parallel, you require a coil to terminate the rung. If you have not provided error evaluation, assign a “dummy” operand to the coil, for example a temporary local data bit.

You can connect shift boxes in series. If the ENO output from the preceding box leads to the EN input of the subsequent box, the subsequent is always processed. If you want to use the result from the preceding box as input value for a subsequent box, variables from the temporary local data area make a convenient intermediate buffer.

If you arrange several boxes in one rung (parallel to the left power rail and then further in series), the boxes in the uppermost branch are processed first from left to right, followed by the boxes in the second branch from left to right, and so on.

You can find further examples of the representation and arrangement of shift functions in the library “Digital Functions” program (FB 111) in the library “LAD_Book” that you can download from the publisher's Website (see page 8).

Shift function in a rung (FBD)

If you want the shift box processed in dependence on specific conditions, you can arrange binary logic operations before the EN input. You can connect the ENO output with binary inputs of other functions; for instance, you can arrange shift boxes in series, whereby the EBO output of the preceding box leads to the EN output of the subsequent box. If you want to use the result from the preceding box as input value to a subsequent box, variables from the temporary local data area make convenient intermediate buffers.

EN and ENO need not be wired.

You can find further examples of the representation and arrangement of shift functions in the program “Digital Functions” (FB 111) in the library “FBD_Book” that you can download from the publisher's Website (see page 8).

13.2 Shift

Shift word variable to the left

The shift function SHL_W shifts the contents of the WORD variable at input IN bit by bit to the left the number of positions specified by the shift number at input N. The bit positions freed up by the shift are padded with zeroes. The WORD variable at output OUT contains the result.

The shift number at input N specifies the number of bit positions by which the contents are to be shifted. It can be a constant or a variable. If the shift number is 0, the function is not executed; if it is greater than 15, the output variable contains zero following execution of the SHL_W function.

Shift doubleword variable to the left

The shift function SHL_DW shifts the contents of the DWORD variable at input IN bit by bit to the left the number of positions specified by the shift number at input N. The bit positions freed up by the shift are padded with zeroes. The DWORD variable at output OUT contains the result.

The shift number at input N specifies the number of bit positions by which the contents are to be shifted. It can be a constant or a variable. If the shift number is 0, the function is not executed; if it is greater than 31, the output variable contains zero following execution of the SHL_DW function.

Shift word variable to the right

The shift function SHR_W shifts the contents of the WORD variable at input IN bit by bit to the right the number of positions specified by the shift number at input N. The bit positions freed up by the shift are padded with zeroes. The WORD variable at output OUT contains the result.

The shift number at input N specifies the number of bit positions by which the contents are to be shifted. It can be a constant or a variable. If the shift number is 0, the function is not executed; if it is greater than 15, the output variable contains zero following execution of the SHR_W function.

Shift doubleword variable to the right

The shift function SHR_DW shifts the contents of the DWORD variable at input IN bit by bit to the right the number of positions specified by the shift number at input N. The bit positions freed up by the shift are filled with zeroes. The DWORD variable at output OUT contains the result.

The shift number at input N specifies the number of bit positions by which the contents are to be shifted. It can be a constant or a variable. If the shift number is 0, the function is not executed; if it is greater than 31, the output variable contains zero following execution of the SHR_DW function.

Shift word variable with sign

The shift function SHR_I shifts the contents of the INT variable at input IN bit by bit to the right the number of positions specified by the shift number at input N. The bit positions freed up by the shift are filled with the signal state of bit 31 (which is the sign of the INT number), that is, with “0” if the number is positive and

with “1” if the number is negative. The INT variable at output OUT contains the result.

The shift number at input N specifies the number of bit positions by which the contents are to be shifted. It can be a constant or a variable. If the shift number is 0, the function is not executed; if it is greater than 15, all bit positions in the output variable contain the sign following execution of the SHR_I function.

For a number in data format INT, shifting to the right is equivalent to division with an exponential number to base 2. The exponent is the shift number. The result of such a division corresponds to the integer rounded down.

Shift doubleword variable with sign

The shift function SHR_DI shifts the contents of the DINT variable at input IN bit by bit to the right the number of positions specified by the shift number at input N. The bit positions freed up by the shift are padded with the signal state of bit 15 (which is the sign of the DINT number), that is, with “0” if the number is positive and with “1” if the number is negative. The DINT variable at output OUT contains the result.

The shift number at input N specifies the number of bit positions by which the contents are to be shifted. It can be a constant or a variable. If the shift number is 0, the function is not executed; if it is greater than 31, all bit positions in the output variable contain the sign following execution of the SHR_DI function.

For a number in data format DINT, shifting to the right is equivalent to division with an exponential number to base 2. The exponent is the shift number. The result of such a division corresponds to the integer rounded down.

13.3 Rotate

Rotate doubleword variable to the left

The shift function ROL_DW shifts the contents of the DWORD variable at input IN bit by bit to the left the number of positions specified in the shift number at input N. The bit positions freed up by the shift are padded with the bit positions that were shifted out. The DWORD variable at output OUT contains the result.

The shift number at input N specifies the number of bit positions by which the contents are to be shifted. It can be a constant or a variable. If the shift number is 0, the function is not executed; if it is 32, the contents of the input variable are retained and the status bits are set. If the shift number is 33, a shift of one position is executed; if it is 34, a shift of two positions is executed, and so on (shifting is carried out modulo 32).

Rotate doubleword variable to the right

The shift function ROR_DW shifts the contents of the DWORD variable at input IN bit by bit to the right the number of positions specified by the shift number at input N. The bit positions freed up by the shift are padded with the bit positions that were shifted out. The DWORD variable at output OUT contains the result.

The shift number at input N specifies the number of bit positions by which the contents are to be shifted. It can be a constant or a variable. If the shift number is 0, the function is not executed; if it is 32, the contents of the input variable are retained and the status bits are set. If the shift number is 33, a shift of one position is executed; if the shift number is 34, a shift of two positions is executed, and so on (shifting is executed modulo 32).

14 Word Logic

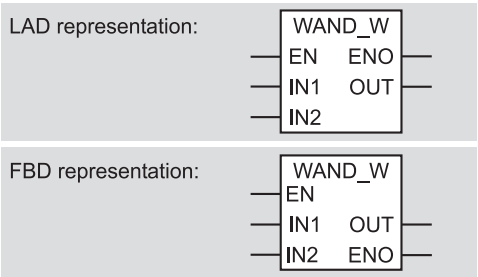
Word logic combines the values of two variables bit by bit according to AND, OR, or Exclusive OR. The logic operation may be applied to words or doublewords. The available word logic operations are listed in Table 14.1.

14.1 Processing a Word Logic Operation

Representation

In addition to the enable input EN and the enable output ENO, the box for a word logic operation has two inputs, IN1 and IN2, and one output, OUT. The “header” in the box identifies the word logic operation executed (for example, WAND_W stands for word ANDing).

Word logic box
(in example: AND operation word by word)



The values to be combined are at inputs IN1 and IN2, and the result of the operation is at output OUT. The inputs and the output have different data types, depending on the operation: WORD for word (16-bit) operations, and DWORD for doubleword (32-bit) operations. The applied variables must be of the same data type as the inputs or the output.

See Chapter 3.5.4, “Elementary Data Types”, for a description of the bits in the various data formats.

Table 14.1 Overview of Word Logic Operations

Word logic operation	With a word variable	doubleword variable
AND	WAND_W	WAND_DW
OR	WOR_W	WOR_DW
Exclusive OR	WXOR_W	WXOR_DW

Function

The word logic operation is executed when “1” is present at the enable input (when power flows through the input EN). If execution of the operation is not enabled (EN = “0”), the operation does not take place and ENO is also “0”.

IF EN == “1” or not wired	
THEN	ELSE
OUT := IN1 Wlog IN2	
ENO := “1”	ENO := “0”

with Wlog as word logic

If the Master Control Relay (MCR) is active, output OUT is set to zero when the word logic operation is executed (EN = “1”). The MCR does not affect the ENO output.

Word logic operations generate a result bit by bit. Bit 0 of input IN1 is combined with bit 0 of input IN2, and the result is stored in bit 0 of output OUT. The same is done with bit 1, bit 2, and so on, up to bits 15 and 31. Table 14.2 shows the result formation for a single bit.

Chapter 15, “Status Bits”, explains how the word logic operations set the status bits.

Examples

Figure 14.1 shows one example for each word logic operation.

In the case of incremental programming, you will find the word logic operations in the Pro-

Table 14.2
Result Formation in Word Logic Operations

Contents of input IN1	0	0	1	1
Contents of input IN2	0	1	0	1
Result with AND	0	0	0	1
Result with OR	0	1	1	1
Result with Exclusive OR	0	1	1	0

gram Elements Catalog (with VIEW → OVER-VIEWS [Ctrl - K] or INSERT → PROGRAM ELE-MENTS) under “Word Logic”.

Word logic in a rung (LAD)

You can arrange contacts in series and in paral-lel before input EN and after output ENO.

The word logic box itself may be placed after a T-branch or in a branch that leads directly to the left power rail. This branch can also have con-

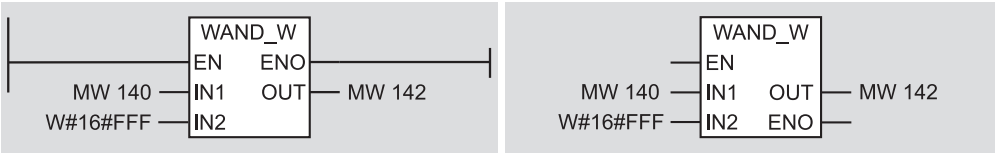
tacts before input EN and it need not be the uppermost branch.

The direct connection to the left power rail allows you to connect word logic boxes in paral-
lel. When connecting boxes in parallel, you require a coil to terminate the rung. If you have not provided error evaluation, assign a “dummy” operand to the coil, for example a temporary local data bit.

You can connect word logic boxes in series. If the ENO output from the preceding box leads to the EN input of the subsequent box, the subse-
quent box is always processed. If you want to use the result of the preceding box as input value for a subsequent box, variables from the temporary local data area make convenient intermediate buffers.

If you arrange several word logic boxes in one rung (parallel to the left power rail and then fur-
ther in series), the boxes in the uppermost branch are processed first from left to right, fol-

AND logic The four high-order bits of memory word MW 140 are set to “0”; the result is stored in memory word MW 142.



OR logic Variables “WLogicVal1” and “WLogicVal2” are ORED bit for bit and the result stored in “WLogicReslt”.



Exclusive-OR logic The value generated by combining variables #Input and #Mask with Exclusive-OR is in variable #Buffer.

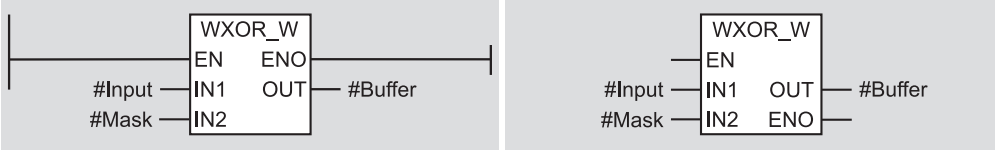


Figure 14.1 Examples of Word Logic Operations

lowed by the boxes in the second branch from left to right, and so on.

You can find further examples of the representation and arrangement of word logic operations in the program “Digital Functions” (FB 111) in the library “LAD_Book” that you can download from the publisher's Website (see page 8).

Word logic in a logic circuit (FBD)

If you want to have the word logic box processed in dependence on specific conditions, you can arrange binary logic operations before the EN input. You can connect the ENO output with binary inputs from other functions; for example, you can arrange word logic boxes in series, whereby the ENO output of the preceding box leads to the EN input of the subsequent box. If you want to use the result from the preceding box as input value for a subsequent box, variables in the temporary local data area make convenient intermediate buffers.

EN and ENO need not be wired.

You can find further examples of the representation and arrangement of logic operations in the program “Digital Functions” (FB 111) in the library “FBD_Book” that you can download from the publisher's Website (see page 8).

14.2 Description of the Word Logic Operations

AND operation

AND combines the individual bits of the value at input IN1 with the corresponding bits of the value at input IN2 according to AND. A bit in result word OUT will be “1” only if the corre-

sponding bit in both values to be combined is “1”.

Since the bits that are “0” at input IN2 also set the corresponding result bits to “0” regardless of what value these bits have at input IN1, we also refer to these bits as being “masked”. Masking is the primary use for the (digital) AND operation.

OR operation

OR combines the individual bits of the value at input IN1 with the corresponding bits of the value at input IN2 according to OR. A bit result word OUT will be “0” only if the corresponding bit in both values to be combined is “0”.

Since the bits that are “1” at input IN2 also set the corresponding result bits to “1” regardless of what value these bits have at input IN1, we also refer to these bits as being “masked”. Masking is the primary use for the (digital) OR operation.

Exclusive OR operation

Exclusive OR combines the individual bits of the value at input IN1 with the corresponding bits of the value at input IN2 according to Exclusive OR. A bit in result word OUT will be “1” only if the corresponding bit in only one of the two values to be combined is “1”. If a bit at input IN2 is “1”, the corresponding bit in the result is the reverse of the bit at the same position in IN1.

In the result, only those bits with opposing signal states in IN1 and IN2 prior to execution of the digital Exclusive OR operation will be “1”. Locating bits with opposing signal states or “negating” the signal states of individual bits is the primary use for the (digital) Exclusive OR operation.

Program Flow Control

LAD and FBD provide you with a variety of options for controlling the flow of the program. You can exit linear program execution within a block or you can structure the program with programmable block calls. You can affect program execution in dependence on values calculated at runtime, or in dependence on process parameters, or in accordance with your plant status.

The **status bits** provide information on the result of an arithmetic or mathematical function and on errors (such as a range violation during a calculation). You can incorporate the signal states of the status bits directly in your program using contacts.

You can use **jump functions** to branch unconditionally or in dependence on the RLO.

A further method of affecting program execution is provided by the **Master Control Relay** (MCR). Originally developed for relay contactor controls, LAD and FBD offer a software version of this program control method.

You can use **block functions** to structure your program. You can use functions and function blocks again and again by defining **block parameters**.

Chapter 19, “Block Parameters”, contains the examples shown in Chapter 5, “Memory Functions”, and Chapter 8, “Counters”, this time programmed as function blocks with block parameters. These function blocks are then also called in the “Feed” example as local instances.

15 Status Bits

Status bits RLO, BR, CC0, CC1 and overflow; setting and evaluating the status bits; using the binary result; EN/ENO

16 Jump Functions

Jump unconditionally; jump in dependence on the RLO

17 Master Control Relay

MCR-dependence; MCR range; MCR zone

18 Block Functions

Block types, block call, block end; status local data; data block register, using data operands; handling data blocks

19 Block Parameters

Parameter declaration; formal parameters, actual parameters; passing parameters to called blocks; Examples: Conveyor belt, parts counter and supply

15 Status Bits

The status bits are binary “flags” (condition code bits). The CPU uses them for controlling the binary logic operations and sets them during digital processing. You can check these status bits or act upon specific bits. The status bits are combined into a word, the status word. However, you cannot access this status word with LAD or FBD.

15.1 Description of the Status Bits

Table 15.1 shows the available status bits. The CPU uses the binary flags for controlling the binary functions; the digital flags indicate mainly results of arithmetic and mathematical functions.

First check

The /FC status bit steers the binary logic within a logic control system. A bit logic step always starts with /FC = “0” and a binary check instruction, the first check. The first check sets /FC = “1”. The first check corresponds in

LAD to the first contact in a network, and in FBD to the first binary function input.

A bit logic step ends with a binary value assignment (e.g. of a single coil or an assignment) or with a conditional jump or a block change. These set /FC = “0”.

Result of the logic operation (RLO)

The RLO status bit is the intermediate buffer in binary logic operations. In the first check, the CPU transfers the check result to the RLO, combines the check result with the stored RLO on each subsequent check, and stores the result, in turn, in the RLO.

You can store the RLO with the SAVE coil/box in the binary result BR. Memory functions, timers and counters are controlled using the RLO and certain jump functions are executed. The RLO corresponds in LAD to the power flowing in the rung (RLO = “1” is the same as “power flowing”).

Status

Status bit STA corresponds to the signal state of the checked binary operand. In the case of memory functions, the value of STA is the same as the written value or (if no write operation takes place, for example if the RLO = “0” or the MCR is active), STA corresponds to the value of the addressed (and unmodified) binary operand.

In the case if edge evaluations FP or FN, the value of the RLO prior to the edge evaluation is stored in STA. All other binary functions set STA = “1”.

The STA status bit has no effect on the processing of the LAD or FBD functions.

Table 15.1 Status Bits

Binary Flags	
/FC	First Check
RLO	Result of logic operation
STA	Status
OR	Status bit OR
BR	Binary result
Digital Flags	
OS	Stored overflow
OV	Overflow
CC0	Condition code (status) bit 0
CC1	Condition code (status) bit 1

Status bit OR

Status bit OR stores the result of a fulfilled series circuit or a fulfilled AND condition and indicates to a subsequently processed parallel circuit or OR function that the result has already been determined. All other binary functions reset the OR status bit.

Overflow

Status bit OV indicates a range violation or the use of invalid REAL numbers. The following functions affect the OV status bit: Arithmetic functions, mathematical functions, some conversion functions, REAL comparison functions.

You can check the OV status bit directly.

Stored overflow

The OS status bit stores a set OV status bit: Whenever the CPU sets status bit OV, it also sets status bit OS. However, while the next properly executed operation resets OV, OS remains set. This provides you with the opportunity of evaluating a range violation or an operation with an invalid REAL number, even at a later point in your program.

You can check status bit OS directly. A block change resets the OS status bit.

Status bits CC0 and CC1 (condition code bits)

Status bits CC0 and CC1 provide information on the result of a comparison function, an arithmetic or mathematical function, a word logic operation, or on the bit shifted out by a shift function.

You can check all combinations of CC0 and CC1 directly (see below).

Binary result

LAD and FBD use status bit BR to implement the EN/ENO mechanism for boxes. You can also set, reset or check status bit BR yourself.

15.2 Setting the Status Bits

The digital functions affect status bits CC0, CC1, OV and OS as shown in Table 15.2. You can check these status bits immediately following the function box.

Status bits in INT and DINT calculations

The arithmetic functions with data formats INT and DINT set all digital status bits. A result of zero sets CC0 and CC1 to "0". CC0 = "0" and CC1 = "1" indicates a positive result, CC0 = "1" and CC1 = "0" indicates a negative result. A range violation sets OV and OS (please note the other meaning of CC0 and CC1 in the case of overflow). Division by zero is indicated by all digital status bits being set to "1".

Status bits in REAL calculations

The arithmetic functions with data format REAL and the mathematical functions set all digital status bits. A result of zero sets CC0 and CC1 to "0". CC0 = "0" and CC1 = "1" indicates a positive result, CC0 = "1" and CC1 = "0" indicates a negative result. A range violation sets OV and OS (please note the other meaning of CC0 and CC1 in the case of overflow). An invalid REAL number is indicated when all digital status bits are set to "1".

A REAL number is referred to as "denormalized" if it is represented with reduced accuracy. The exponent is then zero; the absolute value of a denormalized REAL number is less than $1.175\,494 \times 10^{-38}$. S7-300 CPUs treat denormalized REAL numbers as though they were zero (also see Chapter 3.5.4, "Elementary Data Types").

Status bits for conversion functions

Of the conversion functions, the two's complements affect all digital status bits. In addition, the following conversion functions set status bits OV and OS in the event of an error (range violation or invalid REAL number):

- ▷ I_BCD and DI_BCD:
Conversion of INT and DINT to BCD
- ▷ CEIL, FLOOR, ROUND, TRUNC:
Conversion of REAL to DINT

Tabelle 15.2 Setting the Status Bits

INT calculation				
The result is:	CC0	CC1	OV	OS
< -32 768 (ADD_I, SUB_I)	0	1	1	1
< -32 768(MUL_I)	1	0	1	1
-32 768 to -1	1	0	0	-
0	0	0	0	-
+1 to +32 767	0	1	0	-
> +32 767 (ADD_I, SUB_I)	1	0	1	1
> +32 767(MUL_I)	0	1	1	1
32 768(DIV_I)	0	1	1	1
(-) 65 536	0	0	1	1
Division by zero	1	1	1	1

REAL calculation				
The result is:	CC0	CC1	OV	OS
+ normalized	0	1	0	-
± denormalized	0	0	1	1
± zero	0	0	0	-
- normalized	1	0	0	-
+ infinite (division by zero)	0	1	1	1
- infinite (division by zero)	1	0	1	1
± invalid REAL number	1	1	1	1

Conversion NEG_I				
The result is:	CC0	CC1	OV	OS
+1 to +32 767	0	1	0	-
0	0	0	0	-
-1 to -32 767	1	0	0	-
(-) 32 768	1	0	1	1

Shift function				
The shifted out bit is:	CC0	CC1	OV	OS
“0”	0	0	0	-
“1”	0	1	0	-
with shift number 0	-	-	-	-

DINT calculation				
The result is:	CC0	CC1	OV	OS
< -2 147 483 648 (ADD_DI, SUB_DI)	0	1	1	1
< -2 147 483 648 (MUL_DI)	1	0	1	1
-2 147 483 648 to -1	1	0	0	-
0	0	0	0	-
+1 to +2 147 483 647	0	1	0	-
> +2 147 483 647 (ADD_DI, SUB_DI)	1	0	1	1
> +2 147 483 647 (MUL_DI)	0	1	1	1
2 147 483 648(DIV_DI)	0	1	1	1
(-) 4 294 967 296	0	0	1	1
Division by zero (DIV_DI, MOD_DI)	1	1	1	1

Comparison				
The result is:	CC0	CC1	OV	OS
equal to	0	0	0	-
greater than	0	1	0	-
less than	1	0	0	-
invalid REAL number	1	1	1	1

Conversion NEG_D				
The result is:	CC0	CC1	OV	OS
+1 to +2 147 483 647	0	1	0	-
0	0	0	0	-
-1 to -2 147 483 647	1	0	0	-
(-) 2 147 483 648	1	0	1	1

Word logic				
The result is:	CC0	CC1	OV	OS
zero	0	0	0	-
not zero	0	1	0	-

Status bits for comparison functions

The comparison functions set the CC0 and CC1 status bits. The flags are set independently of the executed comparison function.

Status bits for shift functions

In the case of the shift functions, the signal state of the last bit to be shifted out is transferred to status bit CC1. CC0 and OV are reset.

Status bits for word logic

If the result of the word logic operation is zero (all bits are “0”), CC1 is reset; if at least one bit in the result is “1”, CC1 is set. CC0 and OV are reset.

15.3 Evaluating the Status Bits

LAD: You can use the normally-open (NO) and the normally-closed (NC) contact to check the digital status bits and the binary result. Figure 15.1 shows the check with a normally-open contact. The check with a normally-closed contact returns the negated check result. You can handle the NO and NC contacts for evaluating the status bits in exactly the same way as the “normal” contacts. You can find examples of evaluating the status bits (FB 115 in the “Program Flow Control” program in the “LAD Book” library that you can download from the publisher's Website (see page 8).

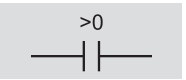
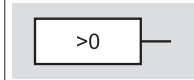
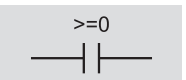
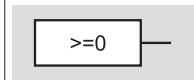

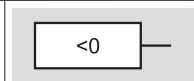
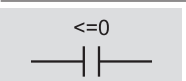
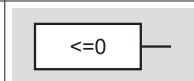
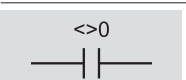
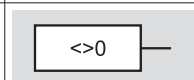
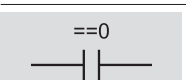
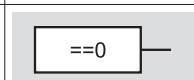

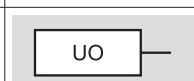



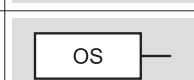

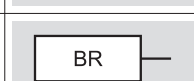
LAD representation	FBD representation	A current flows, or the check is fulfilled when
		Result greater than zero [(CC0=0) & (CC1=1)]
		Result greater than or equal to zero [(CC0=0)]
		Result less than zero [(CC0=1) & (CC1=0)]
		Result less than or equal to zero [(CC1=0)]
		Result not equal to zero [(CC0=0) & (CC1=1) v (CC0=1) & (CC1=0)]
		Result equal to zero [(CC0=0) & (CC1=0)]
		Result invalid (unordered) [(CC0=1) & (CC1=1)]
		Overflow [OV=1]
		Stored overflow [OS=1]
		Binary result [BR=1]

Figure 15.1 Evaluating the Status Bits

FBD: Direct or negated checking of the digital status bits and the binary result is possible. This is shown in Figure 15.1 by direct checking for signal state “1”. The check for signal state “0” returns the negated check result. The checks for evaluation of the status bits can be treated exactly the same as the “normal” checks for binary operands. You can find examples of evaluating the status bits (FB 115 in the program “Program Flow Control”) in the “FBD_Book” library that you can download from the publisher’s Website (see page 8) .

In the case of incremental programming, you will find these checks in the Program Elements Catalog (with VIEW → OVERVIEWS [Ctrl - K] or INSERT → PROGRAM ELEMENTS) under “Status Bits”.

15.4 Using the Binary Result

15.4.1 Setting the Binary Result BR

Saving the binary result



SAVE coil in a rung (LAD)

You store the RLO in the binary result using the SAVE coil. If power flows into the SAVE coil, BR is set, otherwise it is reset. You program the SAVE coil in the same way as a “single” coil without binary operands.

Please note that the SAVE coil does not terminate the logic operation (the /FC status bit is not set to “0”). This means that the logic operation preceding the SAVE coil is also the preceding logic operation for the next network.

The SAVE coil cannot be programmed in conjunction with a T-branch.

SAVE box in a logic circuit (FBD)

With the SAVE box, you save the RLO in the binary result. If the RLO is “1” before the SAVE box, BR is set; otherwise, BR is reset. You program the SAVE box the same way you

program an assign box without binary operands.

Please note that the SAVE box does not terminate a logic operation (status bit /FC is not set to “0”). The next network thus begins with an “open” logic operation.

The SAVE box is not permitted after a T-branch.

Controlling the binary result

LAD and FBD affect even the binary result in order to control the ENO output (Figure 15.2). If enable output ENO is wired, its signal state is the same as that of the BR. In certain cases (“BR corresponds to function”), the LAD or FBD function executed sets the binary result as follows:

- ▷ BR := “1”
for MOVE, for the shift functions and for the word logic operations
- ▷ BR := OV
for the arithmetic and mathematical functions
- ▷ BR := OV or “1”
for the conversion functions
- ▷ BR := BR of the called block in the case of block calls

15.4.2 Main Rung, EN/ENO Mechanism

In programming languages LAD and FBD, many boxes have an enable input EN and an enable output ENO. If the enable input is “1”, the function in the box is processed. When the box is processed correctly, the enable output also has signal state “1”. If an error occurs during processing of a box (for example, overflow during execution of an arithmetic function), ENO is set to “0”. If EN has signal state “0”, ENO is also set to “0”.

These characteristics of EN and ENO can be used in order to connect several boxes together in a chain, with the enable output leading to the enable input of the next box (Figure 15.3). This means, for example, that the entire chain can be “disabled” (no boxes are processed if *input I 1.0* in the example has signal state “0”) or the rest of the chain is no longer processed if one box signals an error.

Is ENO switched ?				NO	
YES					
Is EN switched ?		Is EN switched ?		NO	
YES		NO			
Is EN == "1" ?		Is EN == "1" ?			
YES		NO			
BR corre- sponds to func- tion	BR := "0"	BR corre- sponds to func- tion	BR := "1"	BR := "0"	BR not affected

Figure 15.2 General Schematic for Setting the Binary Result

The input EN and the output ENO are not block parameters but statement sequences that the Program Editor itself generates prior to and following all boxes (also in the case of functions and function blocks). The Program Editor uses the binary result to store the signal state at EN while the block is being processed or to check the error flag from the box.

15.4.3 ENO in the Case of User-written Blocks

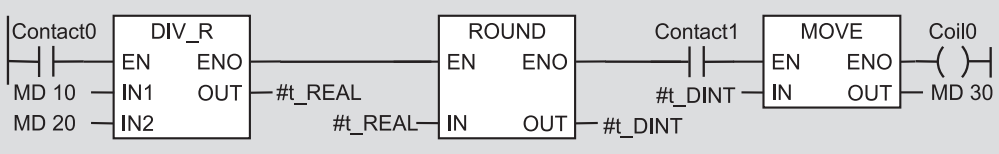
The Program Editor provides the call of your own blocks with the enable input EN and the enable output ENO. You can use the enable input EN to call the block conditionally. You can use the ENO output, for example, to signal a group error (signal state "1", if the block has

been properly processed; "0" if an error has occurred during processing of the block). All system blocks also signal group errors via BR.

You control the ENO output with the binary result BR. The ENO output has the same signal state that BR has when the block is exited.

For example, BR could be set to "1" at the start of a block. If an error then occurs during processing of the block, for example if a result exceeds the fixed range so that further processing must be prevented, set the binary result to "0" with the SAVE coil/box and jump to the end of the block where the block will be exited (in the event of an error, the condition must supply signal state "0"). Please note that the RET coil/box sets the BR to "1" if you exit the block via this coil/box.

Example of a main rung (LAD)



Example of a series connection of boxes (FBD)

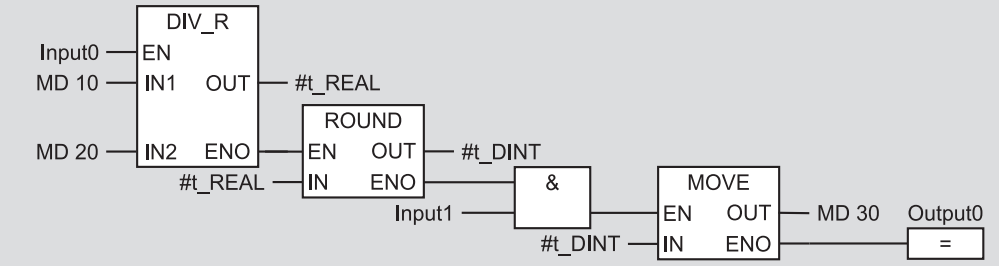


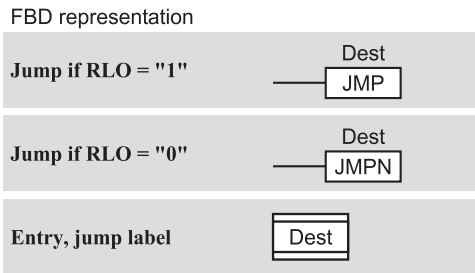
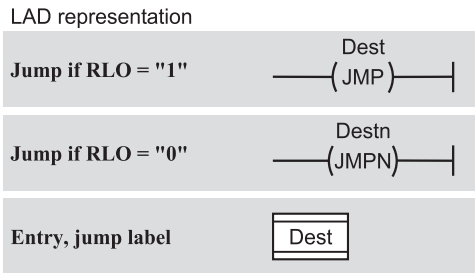
Figure 15.3 Example for the Series Connection of EN and ENO

16 Jump Functions

You can use jump functions to interrupt the linear flow of the program and continue at another point in the block. This program branching can be executed unconditionally or dependent on the RLO.

16.1 Processing a Jump Function

Representation



A jump function consists of the jump operation in the form of a coil (LAD) or box (FBD) and a jump label designating the program location at which processing is to continue after the jump. The jump label is above the jump operation.

Jumps cannot be programmed in conjunction with a T-branch.

A jump label consists of up to 4 characters that can include letters, digits, and the underscore. It begins with letter. A jump label in a box designates the network that is to be processed fol-

lowing completion of the jump operation. The box with the jump label must be at the start of a network ("LABEL" in the Program Elements Catalog).

In the case of incremental programming, you will find the jump functions in the Program Elements Catalog (with VIEW → OVERVIEWS [Ctrl - K] or INSERT → PROGRAM ELEMENTS) under "Jumps".

Function

A jump is either always executed (absolute or unconditional jump) or it is executed depending on the result of the logic operation (RLO) (conditional jump). In the case of a jump dependent on the RLO, you can decide whether the jump is to be executed if the RLO is "1" or if the RLO is "0".

You can execute both forward (in the direction of program processing; in the direction of ascending network numbers) and backward jumps. The jump can take place only within a block, that is, the jump destination must be in the same block as the jump function. If you use the Master Control Relay (MCR), the jump label must be in the same MCR zone or in the same MCR area as the jump function.

The jump destination must be designated unambiguously, that is, you must only assign a jump label once in a block. The jump destination can be jumped to from several locations.

The Program Editor stores the names of the jump labels in the non-executable section of the relevant blocks on the programming device's data medium. Only the widths of the jumps (jump displacement) are stored in the CPU's work memory (in the compiled block). When program modifications are made online to blocks in the CPU, these modifications must therefore always be updated on the program-

ming device's data medium in order to retain the original label names. If this update is not made, or if blocks are transferred from the CPU to the programming device, the non-executable block sections will be overwritten or deleted. On the display or the printout, the Editor then generates replacement symbols for the jump labels (M001, M002 etc.).

16.2 Unconditional Jump

The unconditional jump, which is the jump that is always executed, is jump function JMP,

whose coil is connected to the left power rail (LAD) or whose box has no preceding logic operation (FBD). This jump function is always executed when encountered in the program. The CPU interrupts the linear flow of the program and continues in the network designated by the jump label.

Example (Figure 16.1 and Figure 16.2): In Network 3, there is an unconditional jump to jump label M2. After this network has been processed, the CPU continues program execution at jump label M2 in Network 5. A jump from another program location is required in order to process Network 4.

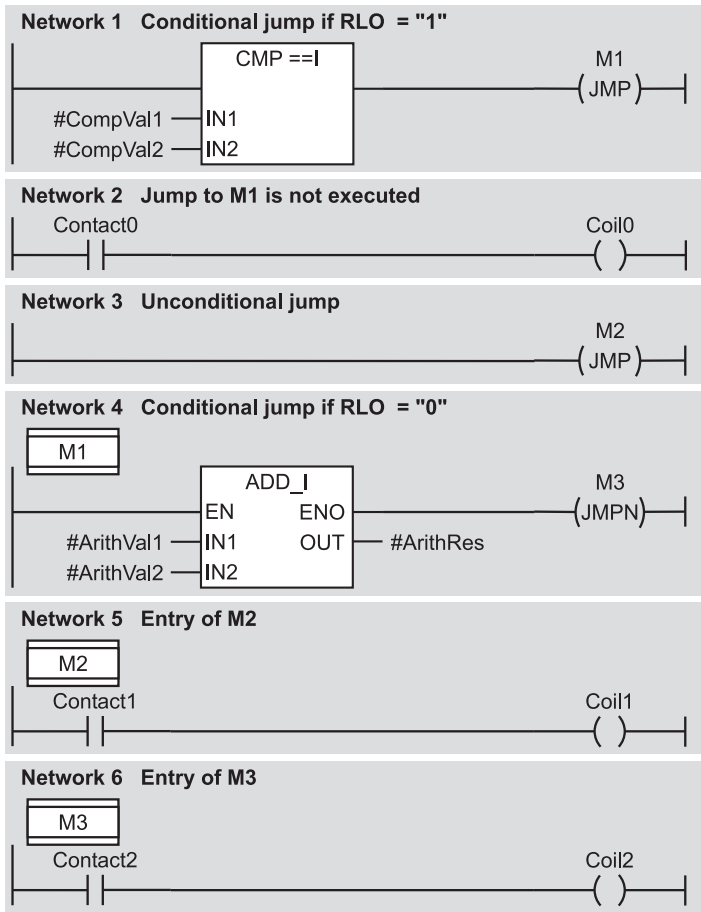


Figure 16.1 Jump Functions Example (LAD)

16.3 Jump if RLO = "1"

The conditional jump if RLO = "1" is jump function JMP, whose coil is not directly connected to the left power rail (LAD) or whose box has a preceding logic operation (FBD). The logic operation preceding this coil can be implemented in any way. If the RLO = "1" (if the preceding logic operation is fulfilled), the CPU interrupts the linear flow of the program and continues in the network designated by the jump label. If the preceding logic operation is not fulfilled, the CPU continues program execution in the next network.

Example (Figure 16.1 and Figure 16.2): If the compare condition in Network 1 is fulfilled, program execution continues in Network 4. If the compare condition is not fulfilled, the next network, which is Network 2, is processed.

16.4 Jump if RLO = "0"

The conditional jump if RLO = "0" is jump function JMPN, whose coil is not directly connected to the left power rail (LAD) or whose box has a preceding logic operation (FBD). The logic operation preceding this coil can be implemented in any way. If the RLO = "0" (if the preceding logic operation is not fulfilled), the CPU interrupts the linear flow of the program and continues in the network designated by the jump label. If the preceding logic operation is fulfilled, the CPU continues program execution in the next network.

Example (Figure 16.1 and Figure 16.2): If the adder in Network 4 signals an error, the jump to Network 6 (jump label M3) is executed. If no error is signaled, Network 5 is the next to be processed.

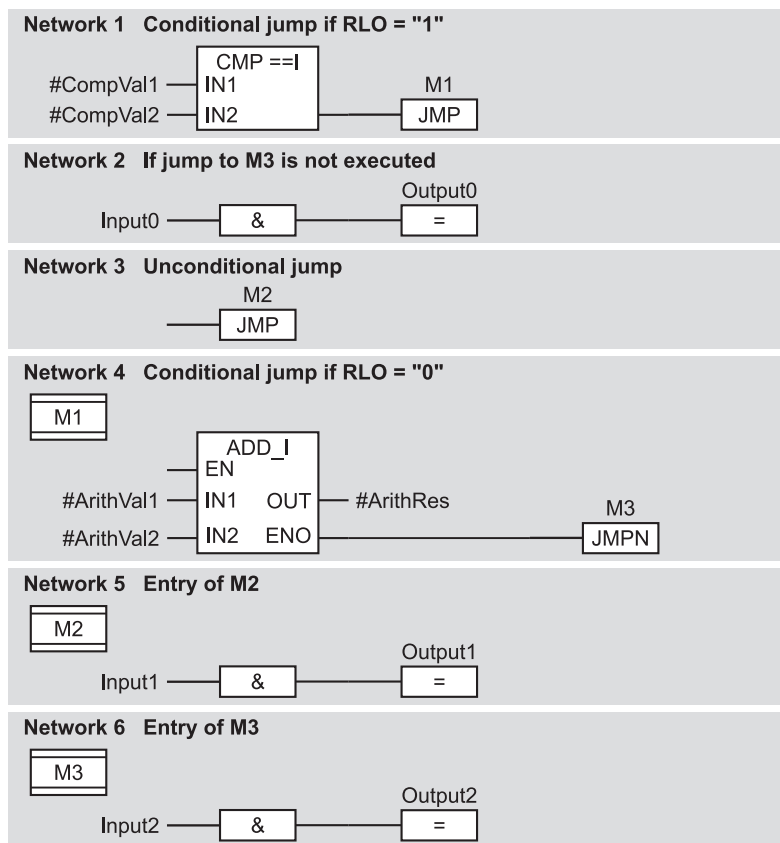


Figure 16.2 Jump Functions Example (FBD)

17 Master Control Relay

In contact control systems, a Master Control Relay activates or deactivates a section of the control system that can consist of one or more rungs.

A deactivated rung

- ▷ deenergizes all non-retentive contactors and
- ▷ retains the states of retentive contactors.

You cannot change the state of the contactors again until the Master Control Relay (MCR) is active.

Please note that deenergizing with the “software” Master Control Relay is no substitute for

an EMERGENCY OFF or safety facility! Treat Master Control Relay switching in exactly the same way as switching with a memory function!

With MCRA and MCRD, you designate an area in your program in which MCR dependency is to take effect. Within this area, you use MCR< and MCR> to define one or more zones in which the MCR dependency can be enabled and disabled. You can also nest the MCR zones. The result of the logic operation (RLO) immediately preceding an MCR zone enables or disables MCR dependency within that zone.

17.1 MCR Dependency


MCR dependency affects coils and boxes. If MCR dependency is enabled (corresponds to Master Control Relay disabled)


- ▷ a single coil or assign box and a midline output set the binary operand to signal state “0” (following the midline output, the RLO is then = “0”, that is, power no longer flows)
- ▷ a set and reset coil or box no longer affect the signal state of the binary operand (“freeze it”)
- ▷ an SR and RS box no longer affect the signal state of the binary operand (“freeze it”)
- ▷ a transfer operation writes zero to the digital operand (every function output of a box of digital data type then writes zero to the operand or to the variable)


Additional to the direct influencing of an operand the RLO is then “0” (power no longer flows) behind a midline output and a T-branch.


Some LAD and FBD functions use transfer statements (invisible to the user). Since a transfer statement writes the value zero if MCR dependency is switched on, the corresponding function can no longer be guaranteed

LAD representation


Activate MCR area 

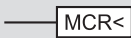
Open MCR zone 


Close MCR zone 

Deactivate MCR area 

FBD representation

Activate MCR area 

Open MCR zone 

Close MCR zone 

Deactivate MCR area 

You must exclude the following program sections from MCR dependency otherwise the CPU will go to STOP or undefined runtime behavior can occur:

- ▷ Block calls with block parameters
- ▷ Accesses to block parameters that are parameter types (e.g. BLOCK_DB)
- ▷ Accesses to block parameters that are components or elements of complex data types or UDTs

You enable MCR dependency in a zone if the RLO is “0” immediately prior to opening the zone (analogous to disabling the Master Control Relay). If you open an MCR zone with RLO “1” (Master Control Relay enabled), processing within this MCR zone takes place without MCR dependency. MCR dependency is effective only within an MCR zone.

In the case of incremental programming, you will find the MCR functions in the Program Elements Catalog (with VIEW → OVERVIEWS

[Ctrl - K] or INSERT → PROGRAM ELEMENTS) under “Program Control”.

17.2 MCR Area

To be able to use the characteristic features of the Master Control Relay, define an MCR area with MCRA (start) and MCRD (end of the MCR area). MCR dependency is active within an MCR area, but not yet enabled (Figure 17.1).

The MCRA coil/box and the MCRD coil/box always stand alone in separate networks.

If you call a block within an MCR area, MCR dependency is deactivated in the block called. An MCR area only starts again with the MCRA coil/box. When a block is exited, MCR dependency is set as it was before the block was called, regardless of the MCR dependency with which the called block was exited.

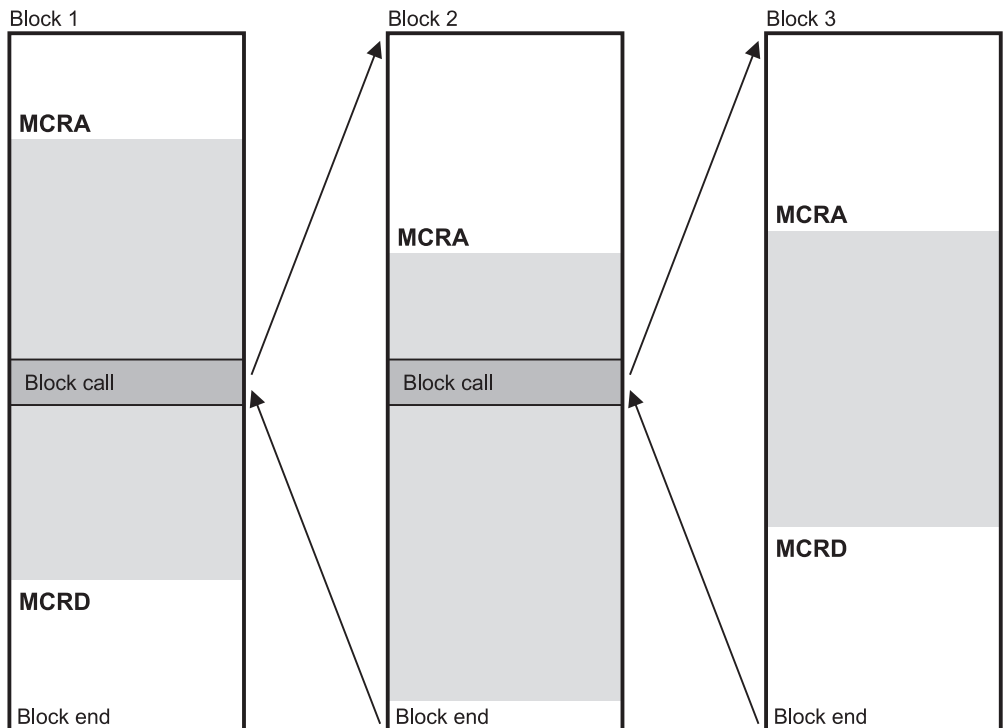


Figure 17.1 MCR Area in the Case of Block Change

17.3 MCR Zone

LAD: You define an MCR zone with the MCR< coil (start) and the MCR> coil (end of the MCR zone). The MCR< coil requires a preceding logic operation; the MCR> coil is connected directly to the left power rail. Both coils terminate a rung. Within this zone, you control MCR dependency with the RLO prior to the MCR< coil: If power flows into the coil, MCR dependency is disabled (“normal” processing); if power does not flow into the coil, MCR dependency is enabled.

FBD: You define an MCR zone with the MCR< box at the beginning and the MCR> box at the end of the MCR zone. The MCR< box requires a preceding logic operation; the MCR> box stands alone in a network. Within this zone, you control MCR dependency with the RLO preceding the MCR< box: If it is “1”, MCR dependency is disabled (“normal” processing); if it is “0”, MCR dependency is enabled.

You can open an MCR zone within another MCR zone. The nesting depth for MCR zones is 8; that is, you can open a zone up to eight times before you close a zone (Figure 17.2).

You control the MCR dependency of a nested MCR zone with the RLO on opening the zone. However, if MCR dependency is enabled in a “supraordinate” zone, you cannot disable MCR dependency in a “subordinate” MCR zone. The Master Control Relay of the first MCR zone controls the MCR dependency in all nested.

A block call within an MCR zone does not change the nesting depth of an MCR zone. The program in the called block is still in the MCR zone that was open when the block was called (and is controlled from here). However, you must reactivate MCR dependency in a called block by opening the MCR area.

In Figure 17.3, memory bits M 10.0 and M 11.0 control the MCR dependencies. With memory bit M 10.0, you can enable MCR dependency in both zones (with “0”) regard-

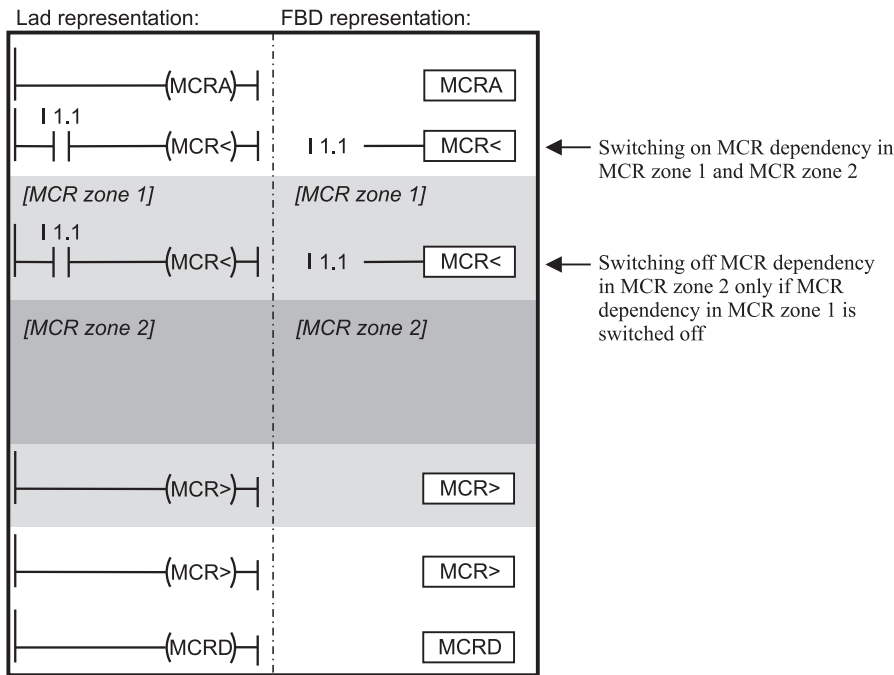


Figure 17.2 MCR Dependency in Nested MCR Zones

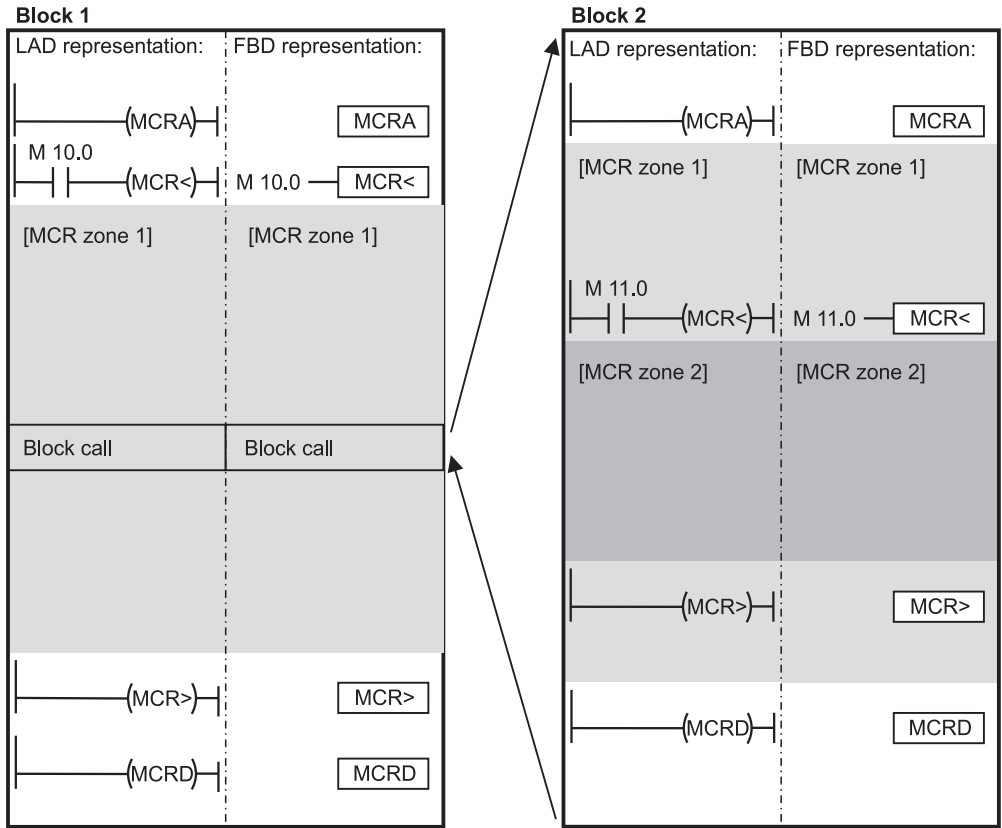


Figure 17.3 MCR Zones in the Case of Block Change

less of the signal state of memory bit M 11.0. If the MCR dependency for zone 1 is disabled with M 10.0 = “1”, you can control the MCR dependency of zone 2 with memory bit M 11.0 (Table 17.1).

17.4 Setting and Resetting I/O Bits

Despite enabled MCR dependency, you can set or reset the bits of an I/O area with the system functions. A requirement for this is that the bits

to be controlled are in the process-image output table or a process-image output table has been defined for the I/O area to be controlled.

The system function **SFC 79 SET** is available for setting and **SFC 80 RSET** for resetting the I/O bits (Table 17.2). You call these system functions in an MCR zone. The system functions are only effective if MCR dependency is enabled; if MCR dependency is disabled, calling these SFCs has no effect.

Table 17.1 MCR Dependency in the Case of Nested MCR Zones (Example)

M 10.0	M 11.0	Zone 1	Zone 2
“1”	“1”	No MCR dependency	
“1”	“0”	No MCR dependency	MCR dependency enabled
“0”	“1” or “0”	MCR dependency enabled	

Table 17.2 Parameters of the SFCs for Controlling the I/O Bits

SFC	Parameter	Declaration	Data Type	Assignment, Description
79	N	INPUT	INT	Number of bits to be set
	RET_VAL	OUTPUT	INT	Error information
	SA	OUTPUT	POINTER	Pointer to the first bit to be set
80	N	INPUT	INT	Number of bits to be reset
	RET_VAL	OUTPUT	INT	Error information
	SA	OUTPUT	POINTER	Pointer to the first bit to be reset

Setting and resetting the I/O bits also simultaneously updates the process-image output table. The I/O are affected byte by byte. The bits not selected with the SFCs (in the first and in the last byte) retain the signal states as they are currently available in the process image.

You can find examples of the Master Control Relay and of the system functions SFC 79 and SFC 80 in function block FB 117 in the libraries “LAD_Book” and “FBD_Book” in the program “Program Flow Control” that you can download from the publisher's Website (see page 8).

18 Block Functions

In this chapter, you will learn how to call and terminate code blocks and how to work with operands from data blocks. The next chapter then deals with using block parameters.

18.1 Block Functions for Code Blocks

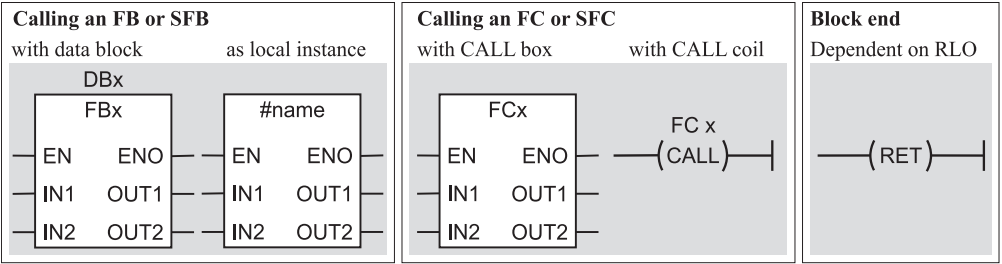
Block functions for code blocks include instructions for calling and terminating blocks (Figure 18.1). Code blocks are called with the call box. If functions or system functions have no block parameters, they can also be called with the CALL coil or with the CALL box. In both cases, a preceding logic operation enabling a conditional call (call dependent on

conditions) is permissible. The block end function RET always requires a preceding logic operation.

In addition to the block change, the call box also contains the transfer of block parameters. When function blocks are called, it also opens the instance data block. The CALL coil/box is no more than a change to another block and is only meaningful (and permissible) in the case of functions and system functions.

After a block has been terminated, and following the call function, the CPU continues program execution in the block that made the call (the calling block). If an organization block is terminated, the CPU returns to the operating system.

LAD representation:



FBD representation:

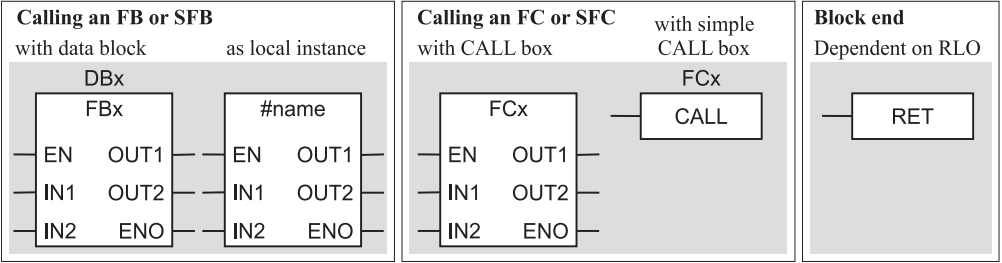


Figure 18.1 Block Functions for Code Blocks

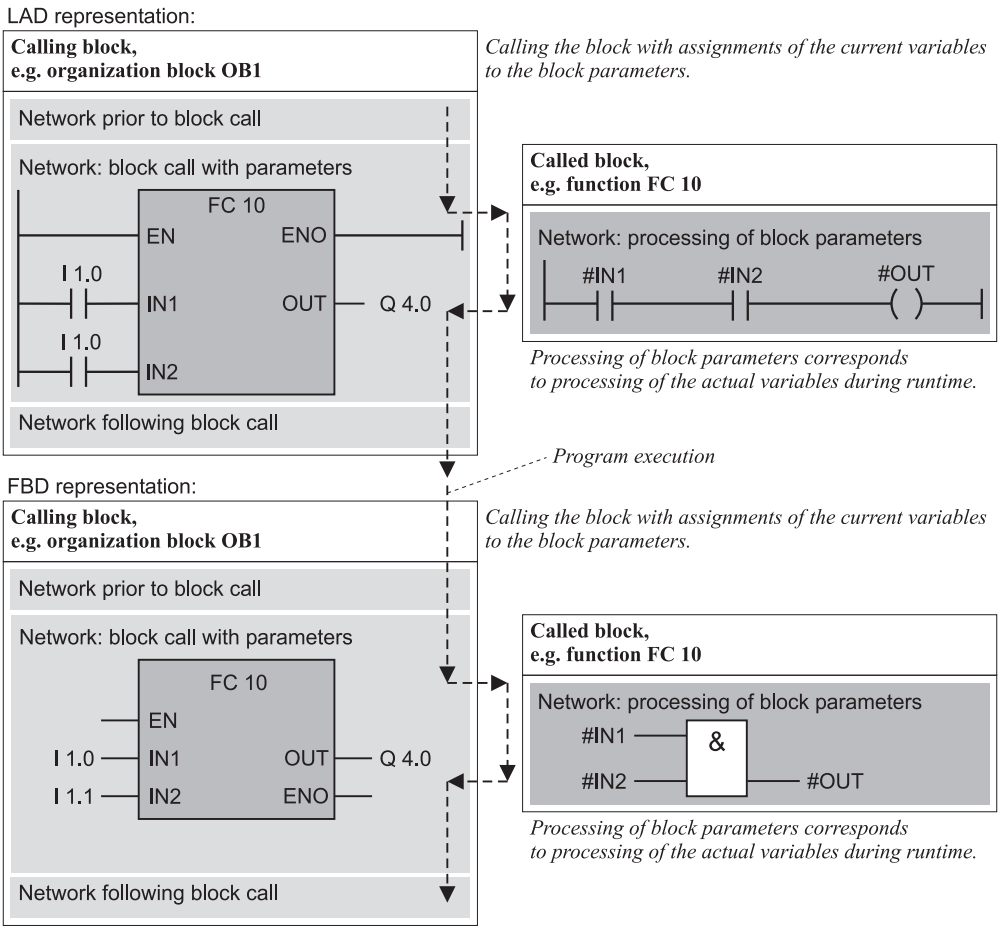
In the case of incremental programming, you will find the CALL coil/box and the RET coil/box in the Program Elements Catalog (with VIEW → OVERVIEWS [Ctrl - K] or INSERT → PROGRAM ELEMENTS) under “Program Control”; you insert block calls with call boxes into your program when you select blocks from “FC/FB/SFC/SFB blocks”, “Multiple Instances” or “Libraries”.

18.1.1 Block Calls: General

If a code block is to be processed, it must be “called”. Figure 18.2 gives an example for calling function FC 10 in organization block OB 1.

A block call consists of the call box that contains the address of the called block (here: FC 10), the enable input EN, the enable output ENO, and any block parameters. Following processing of the call function, the CPU continues program execution in the called block. The block is processed to the end or until a block end function is encountered. The CPU then returns to the calling block (here: OB 1) and continues processing this block after the call box.

The information the CPU requires to find its way back to the calling block is stored in the block stack (B stack). With every new block call, a new stack element is created that



includes the return address, the contents of the data block register and the address of the local data stack of the calling block. If the CPU goes to the Stop state as a result of an error, you can use the programming device to see from the contents of the B stack which blocks were processed up to the error.

You can transfer data to and from the called block for processing. These data are transferred via block parameters. With the call box, you can also call blocks without block parameters.

18.1.2 Call Box

You use the call box to call FBs, FCs, SFBs and SFCs. (You cannot call organization blocks since they are event-driven and are started by the operating system.)

You can use the EN input to make the block call subject to conditions. If the EN input is connected direct to the left power rail, the call is absolute; it is always executed. If there is a logic operation preceding EN, the block call is only executed if the preceding logic operation is fulfilled. The ENO output has the same signal state as the binary result BR on exiting the called block.

IF EN == “1” or not connected		
THEN		ELSE
Called block is processed		Called block is not processed
IF called block returns BR = “1”		
THEN	ELSE	
ENO := “1”	ENO := “0”	
		ENO := “0”

You label the parameters of the called block with the absolute or symbolic operand current for the call. If a parameter is of data type BOOL, precede this parameter with

- ▷ a contact or a rung (LAD) or
- ▷ a binary variable or a binary logic operation (FBD).

A Boolean output parameter cannot be combined further.

LAD: You can arrange several call boxes in series, connecting them with each other via EN

or ENO. You can only insert a call box in a parallel rung if it is connected directly to the left power rail.

FBD: You can connect call boxes in series by connecting the ENO output of one box with the EN input of the next. The ENO outputs of several boxes can be combined with AND or OR.

MCR dependency is de-activated when a block is called. The MCR is disabled in the called block regardless of whether the MCR was enabled or disabled prior to the block call. When exiting a block, MCR dependency assumes the same setting that it had prior to the block call.

Depending on the block parameters, you can modify the contents of the data block registers when the block change is made. If the *called* block is a function block, the instance block is always opened in this block via the DI register. If the *calling* block is a function block, the contents of the DI register (the instance data block) are retained after the block call. The contents of the DB register depend, among other things, on the block parameters that were passed.

Calling function blocks

You call a function block by selecting the relevant function block from the Program Elements Catalog under "FB Blocks". Prerequisite is that the function block to be called must already be in the user program. You write the instance data block belonging to the call above the box. Both blocks (function block and instance data block) can have absolute or symbolic addresses.

In the case of function blocks, you do not need to initialize all block parameters at the call. The uninitialized block parameters retain their current value. However, block parameters saved as pointers should at least be initialized when called for the first time so that meaningful values are entered here (see Chapter 19.3, "Actual Parameters").

You can also call "function blocks with multiple instance capability" within other "function blocks with multiple instance capability" as local instance. In doing so, the called function block uses the instance data block of the calling function block as the store for its local data. Prior to the call, you declare the local instance in the static local data of the calling function

block (the block you are currently programming). The local instance is called by selecting one of the available local instances under 'Multiple Instances' in the Program Elements Catalog; it is not necessary to specify an instance data block (see also Chapter 18.1.6, "Static Local Data").

Calling functions

You call a function by selecting the relevant function under "FC Blocks" in the Program Elements Catalog. The function can have an absolute or a symbolic address.

When you call functions, you must initialize all available parameters.

Calling functions with a function value takes exactly the same form as calling functions with no function value. Only the first output parameter, corresponding to the function value, has the name `RET_VAL`.

Calling system blocks

The CPU operating system contains system functions (SFCs) and system function blocks (SFBs) that you can use. The number and type of system blocks depends on the CPU. You can call all system blocks with the call box.

You call a system function block in the same way as one you have written yourself; set up the associated instance data block in the work memory with the same data type as the SFB.

You call a system function in the same way as a function you have written yourself.

System blocks exist only in the CPU operating system. If you want to call system blocks during offline programming, the Program Editor needs a description of the call interface so it can initialize the parameters. You will find this interface description under *System Function Blocks* on the library named *Standard Library*. From here, the Program Editor copies the interface description to the offline block container when you call a system block. The interface description thus copied then appears as "normal" block object.

The Program Elements Catalog provides the system blocks currently available in the user program under "SFC Blocks" or "SFB Blocks".

You can, for example, select a system block from the Program Elements Catalog with the mouse and drag it to the block currently being processed block, where it is then called. At the same time, this block (or, more precisely: its interface description) is copied into the block container.

Block calls in sequence (FBD)

If you connect block boxes in sequence in FBD, and have to "pass on" binary signals from one box to the next, you must observe the processing sequence of the binary logic operations by the program editor. The binary operands and logic operations are initially processed prior to the box inputs, starting with the "last" block box; the intermediate binary results are saved in the temporary local data (not shown). Following this, the program editor processes the boxes themselves and their binary outputs, commencing with the "first" box.

Figure 18.3 clarifies the processing sequence. The OR box with the three inputs is initially processed prior to the right block, then the OR box with the two inputs, followed by the left block call (control of Pump1) and finally the right block call (Pump2). The temporary local variable `#_P1_Fault` would therefore be scanned first (on the OR box with undefined signal state) and then set (at the binary output of the first block call). This could result in falsification of the result of the logic operation.

Remedy: use a global variable (e.g. a bit memory or a data bit; in this case, a change in status at the output of the first block would only be considered in the next program cycle at the input of the second block) for passing on the signal, or divide the block calls between two networks.

18.1.3 CALL Coil/Box

You can call functions and system functions using the CALL coil/box. It is a requirement that the called blocks have no block parameters. You can use the CALL coil/box if a block is too long or not clear enough for you by simply "breaking down" the block into sections and calling the sections one after the other. One single CALL coil or box is permitted per network.

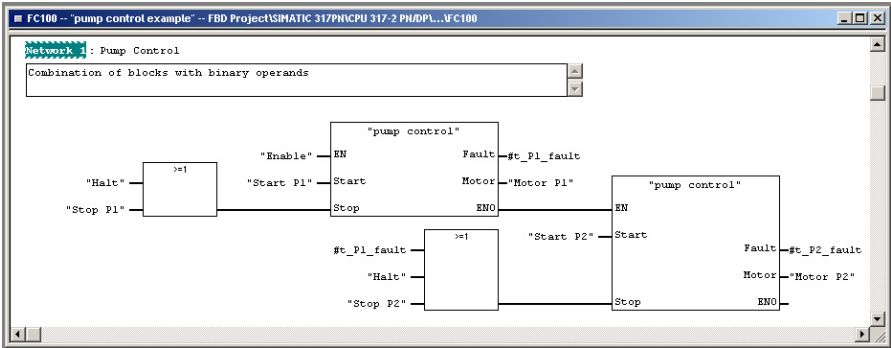


Figure 18.3 Block Calls in Sequence (FBD)

LAD: If the CALL coil is connected directly to the left power rail, the call is always executed (unconditional call). If there is a logic operation preceding the CALL coil, the call is only executed if the preceding logic operation is fulfilled, that is, if power flows into the CALL coil. If the preceding logic operation is not fulfilled, the call is not executed and the next network is processed.

FBD: If there is a logic operation preceding the CALL box, the call is executed only when the preceding logic operation is fulfilled, that is, when RLO = "1" is present at the CALL box. If the preceding logic operation is not fulfilled, the call is not executed and the next network is immediately processed.

When a block change is made, status bit OS is reset; status bits CC0, CC1 and OV are not affected.

MCR dependency is deactivated when a block is called. The MCR is disabled in the called block regardless of whether the MCR was enabled or disabled prior to the block call. When a block is exited, MCR dependency assumes the same setting it had prior to the block call.

Calling a block with the CALL coil/box saves the data block registers in the B stack; the block end restores their contents when the called block is exited. The global data block current prior to the block call and the instance data block are also open following the block call. If no data block was open prior to the block call (for example, no instance data block in OB 1),

no data block is open following the block call either, regardless of which data blocks may be open in the called block.

18.1.4 Block End Function

You can terminate processing in a block prematurely with the block end function RET.

Conditional block end

LAD representation:

FBD representation:

The block end function is represented as a coil or box requiring a preceding logic operation. The RET coil/box must be alone in a network.

If the preceding logic operation is fulfilled, the block is exited. A return jump is made to the previously processed block in which the block call took place. If an organization block is terminated, the CPU returns to the system program.

If the preceding logic operation is not fulfilled, the next network in the block is processed.

IF preceding logic operation == "1"	
THEN	ELSE
The block is exited	The next network is processed
BR := "1"	BR := "0"

The RET coil/box simultaneously stores the RLO (whether power flows or not) in binary result BR, regardless of whether or not the logic operation was fulfilled. The binary result is decisive for controlling the ENO output at the call box (see also Chapter 15, “Status Bits”).

18.1.5 Temporary Local Data

You use the temporary local data to buffer results arising during processing of a block. Temporary local data are available only during block processing; once a block has been processed, your buffered data are lost.

Temporary local data are operands which lie in the local data stack (L stack) in system memory. The CPU’s operating system makes the temporary local data for a code block available when that code block is called. When a block is called, the values in the L stack are virtually coincidental. In order to be able to make sensible use of the local data, you must first write them prior to reading. When the block is terminated, the L stack is assigned to the next block called.

The number of temporary local data bytes a block requires is in the block header. Reading the header tells the operating system how many bytes have to be reserved in the L stack when the block is called. You, too, can tell from the entry in the block header just how many local data bytes the block requires (using the Editor, with the block open, by invoking FILE → PROPERTIES or in the SIMATIC Manager with EDIT → OBJECT PROPERTIES, in each case on tab “General – Part 2”).

Declaring temporary local data

You declare the temporary local data in the declaration section of the code block:

- ▷ under “temp” in the case of incremental programming, or
- ▷ between VAR_TEMP and END_VAR in the case of source-oriented programming.

Figure 18.5 shows an example for the declaration of temporary local data. The variable *difference* lies in the temporary local data and is of data type INT; the variable *buffer* is of data type REAL.

The temporary local data are stored in the L stack order of their declaration in accordance with their data type.

Symbolic addressing of temporary local data

You reference temporary local data using their symbolic names. You assign these names in accordance with the rules for block-local symbols.

All operations allowed for bit memory are also allowed for temporary local data. Please note, however, that a temporary local data bit is not suitable for use as an edge memory bit because it does not retain its signal state outside the relevant block.

You can address the temporary local data for a block only within that block (exception: the temporary local data for the calling block can be accessed via block parameters.)

Size of the L stack

The total size of the L stack is CPU-specific. The available number of temporary local data bytes in a priority class, that is, in the program of an organization block, is also predetermined. On an S7-300, the number of temporary local data bytes is fixed, for example there are 510 bytes per priority class on the CPU 314. On an S7-400, you can specify the number of temporary local data bytes you will need when you initialize the CPU. These bytes must be shared by the blocks called in the relevant organization block as well as by the blocks which they call.

Please note that the Editor also uses temporary local data, for instance for passing block parameters, a fact which goes unnoticed on the programming interface.

Declaration	Name	Data Type	Address
TEMP	SINFO	ARRAY [1..20]	0.0
		BYTE	*1.0
	LByte	ARRAY [1..16]	20.0
		BYTE	*1.0

Figure 18.4

Example of the Declaration of Temporary Local Data in an Organization Block

Start information

When an organization block is called, the CPU operating system passes start information in the temporary local data. This start information comprises 20 bytes for every organization block, and is nearly identical for all OBs. The start information for the various organization blocks is described in detail in Chapters 20, “Main Program”, 21, “Interrupt Handling”, 22, “Start-up Characteristics” and 23, “Error Handling”.

These 20 bytes of information must always be available in each priority class used. If you program a routine for the evaluation of synchronization errors (programming and access errors), you must set aside an additional 20 bytes at least for the start information of these error organization blocks, as these error OBs belong to the same priority class.

You declare the start information for an organization block when you program that block. The information is mandatory. Sample declarations in English can be found on the *Standard Library* under *Organization Blocks*. If you do not need the start information, simply declare the first 20 bytes as something else, for example as an array (as shown in Figure 18.5).

Absolute addressing of temporary local data

Normally, you reference temporary local data by their symbolic names. The use of absolute addresses is the exception. Once you are familiar with the way data are stored in the L stack, you can compute the addresses of the static local data yourself. You will also see the

addresses listed in the variable declaration table of the compiled block.

The operand identifier for temporary local data is L; a bit is addressed with L, a byte with LB, a word with LW, and a doubleword with LD.

Example: You want to reserve 16 bytes of temporary local data for absolute addressing, and you want to reference the values in these bytes by both byte and bit. To do this, create an array at the beginning of the local data area so that addressing begins at 0. In an organization block, you would place this array declaration immediately behind the declaration for the start information, in which case addressing would begin at 20.

Data type ANY

A variable in the temporary local data can – although this is an exception – be declared as data type ANY. You can use this feature to modify the ANY pointer at runtime (see Chapter 24.2.5, ““Variable” ANY Pointer”).

18.1.6 Static Local Data

Static local data are operands which a function block stores in its instance data block.

Static local data are a function block's “memory”. They retain their values until those values are changed by the program, just like data operands in global data blocks.

The number of static local data bytes is limited by the data types of the variables and by the CPU-specific length of a data block.

Declaration	Name	Data Type	Address	Initial Value	Comment
IN	Man_on	BOOL	0.0	FALSE	Input parameter
OUT	Switch_on	BOOL	2.0	FALSE	Output parameter
IN_OUT	Length	INT	4.0	0	I/O parameter
STAT	Total Setpoint	INT	6.0	0	Static local data
		DINT	8.0	L#0	
TEMP	Difference Buffer	INT	0.0		Temporary local data
		REAL	2.0		

Figure 18.5 Example of the Declaration of Local Data in a Function Block

Declaring static local data

You declare static local data in the declaration section of the function block:

- ▷ under “stat” in the case of incremental programming or
- ▷ between VAR and END_VAR in the case of source-oriented programming.

Figure 18.5 in Chapter 18.1.5, “Temporary Local Data” shows an example of a variable declaration in a function block. The block parameters are declared first, then the static local data, and finally the temporary local data.

The static local data are stored in the instance data block behind the block parameters in the order of their declarations and in accordance with their data types.

Symbolic addressing of static local data

You reference static local data with symbolic names. You assign these names in accordance with the rules for block-local symbols.

All the same operations that can address data operands in global data blocks can also address static local data.

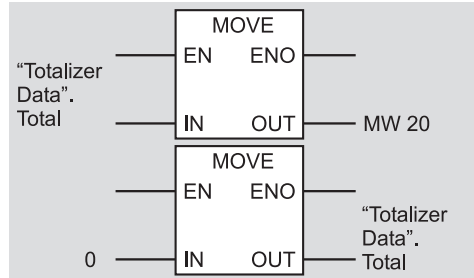
Example: The function block “Totalizer” adds an input value to a value stored in the static local data and then stores the total in the static local data. At the next call, the input value is again added to this total, and so on (Figure 18.6 above).

Total is a variable in the data block “Totalizer-Data”, the instance data block for the “Totalizer” function block (you can define the names of all blocks yourself in the Symbol Table, but you must stick to the applicable rules when doing so). The instance data block has the data structure of the function block; in the example, it contains two INT variables with the names *In* and *Total*.

Accessing static data outside the function block

As a rule, the static local data are processed only in the function block itself. Because they are stored in a data block, however, you can access the static local data at any time with “Data Block Name”.Operand Name just as you would a variable in a global data block.

In our little example, the data block is named *TotalizerData* and the data operand is named *Total*. The applicable access instructions might be as follows:



Local instances

When you call a function block, you normally specify the instance data block for that call. The function block then stores its block parameters and its static local data in this instance data block.

Beginning STEP 7 V2, you can generate “multiple instances”, that is, you can call a function block as a local instance in another function block. The static local data (and the block parameters) of the function block called are then a subset of the calling block's static local data. Prerequisite is that both the calling function block and the called function block have block version 2, that is, that both have “multiple instance capability”. This allows you to “nest” function block calls up to a depth of eight.

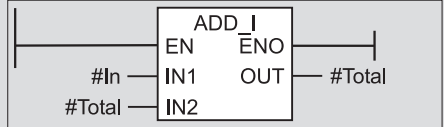
Example (Figure 18.6 below): In the static local data of the function block “Evaluation”, you declare a variable *Memory* that corresponds to the function block “Totalizer” and has its data structure. Now you can call the function block “Totalizer” via the variable *Memory*, but without specifying a data block because the data for *Memory* are stored “block-locally” in the static local data (*Memory* is the local instance of the function block “Totalizer”).

You access *Memory*'s static local data in the program in function block “Evaluation” the same way you would access structure components, which is by specifying the structure name (*Memory*) and the component name (*Total*).

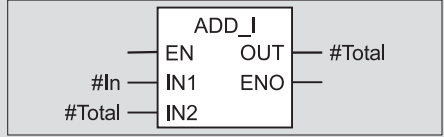
FB "Totalizer"

Address	Declaration	Name	Type
+ 0.0	in	In	INT
+ 2.0	stat	Total	INT

LAD representation:



FBD representation:



DB "TotalizerData"

Address	Declaration	Name	Type
+ 0.0	in	In	INT
+ 2.0	stat	Total	INT

In the Data view, the data block shows all individual variables so that the variables of a local instance appear with their full names.

Simultaneously, you see the corresponding absolute addresses.

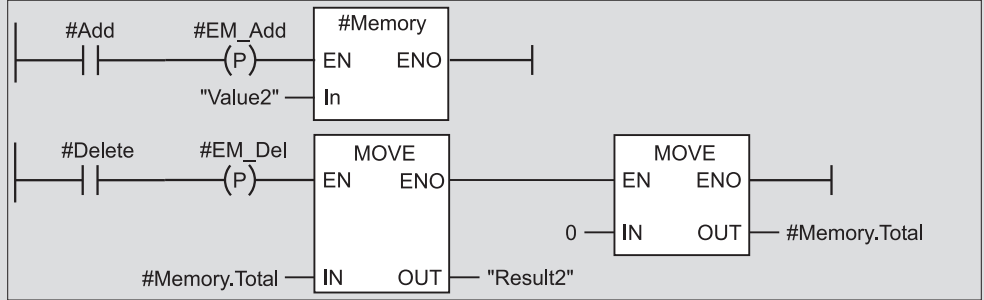
FB "Evaluation"

Address	Declaration	Name	Type
0.0	in	Add	BOOL
0.1	in	Delete	BOOL
2.0	stat	EM_Add	BOOL
2.1	stat	EM_Del	BOOL
4.0	stat	Memory	Totalizer

DB "EvaluationData"

Address	Declaration	Name	Type
0.0	in	Add	BOOL
0.1	in	Delete	BOOL
2.0	stat	EM_Add	BOOL
2.1	stat	EM_Del	BOOL
4.0	stat:in	Memory.In	INT
6.0	stat	Memory.Total	INT

LAD representation:



FBD representation:

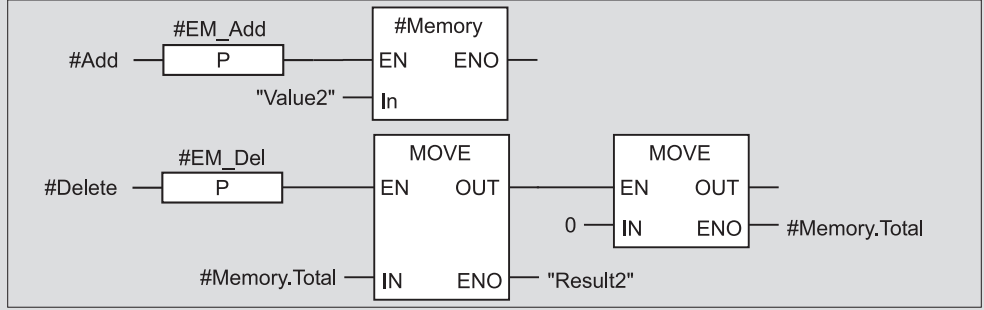


Figure 18.6 Example of Static Local Data and Local Instances

The instance data block “EvaluationData” thus contains the variables *Memory.In* and *Memory.Total*, which you can also address as global variables, for example as “*Evaluation-Data*” *Memory.Total*.

You can find this example of the use of a local instance in function blocks FB 6, FB 7 and FB 8 in program “Program Flow Control” in the libraries “LAD_Book” and “FBD_Book”, that you can download from the publisher's Website (see page 8). The “Feed” example in Chapter 19.5.3, “Feed Example” contains additional usage of local instances.

Absolute addressing of static local data

As a rule, the static local data are addressed symbolically. Absolute addressing is the exception. Within a function block, the instance data block is opened via the DI register. Operands in this data block, and this includes the static local data as well as the block parameters, therefore have DI as operand identifier. A bit is addressed with DIX, a byte with DIB, a word with DIW, and a doubleword with DID.

Once you know just how data are stored in a data block, you can compute the absolute addresses of the static local data yourself. You can also find the addresses in the compiled block in the variable declaration table. But be very careful! *These addresses are addresses that are relative to the start of the instance.* They apply only when you call the function block with a data block. If you call the function block as local instance, the local instance's local data are located right in the middle of the calling function block's instance data block. You can view the absolute addresses, for example, in the compiled instance data block which contains all local instances. Select VIEW → DATA VIEW when you want to read up on the addresses of individual local data operands.

Using our example as a basis, the variable *Total* in the function block “Totalizer” could be addressed with DIW 2 if function block “Totalizer” were called with a data block (also see the operands in the data block “TotalizerData”), and with DIW 6 if function block “Totalizer” were called as local instance in function block “Evaluation” (also see the operands in data block “EvaluationData”).

18.2 Block Functions for Data Blocks

You store your program data in the data blocks. In principle, you can also use the bit memory area for storing data; however, with the data blocks, you have significantly more possibilities with regard to data volume, data structuring and data types. This chapter shows you

- ▷ how to work with data operands,
- ▷ how to call data blocks and
- ▷ how to create, delete and test data blocks at runtime.

You can use data blocks in two versions: as *global data blocks* that are not assigned to any code block, and as *instance data blocks* that are assigned to a function block. The data in the global data blocks are “free” data that every code block can make use of. You yourself determine their volume and structure directly through programming the global data block. An instance data block contains only the data with which the associated function block works; this function block then also determines the structure and storage location of the data in “its” instance data block.

The number and length of data blocks are CPU-specific. The numbering of the data block begins at 1; there is no data block DB 0. You can use each data block either as a global data block or as an instance data block.

You must first create (“set up”) the data blocks you use in your program, either by programming, such as code blocks, or at runtime using the system function SFC 22 CREAT_DB.

18.2.1 Two Data Block Registers

Each S7-CPU has two data block registers. These registers contain the numbers of the current data blocks; these are the data blocks with whose operands processing is currently taking place. Before accessing a data block operand, you must open the data block containing the operand. If you use fully-addressed access to data operands (with specification of the data block, see below), you need not be concerned with opening the data blocks or with the contents of the data block registers. The Editor gen-

erates the necessary instructions from your specifications.

The Program Editor uses the first data block register preferably for accessing global data blocks and the second data block register for accessing instance data blocks. For this reason, these registers are given the names “Global data block register” (DB register for short) and “Instance Data Block Register” (DI register for short). The handling of the registers by the CPU is absolutely identical. Each data block can be opened via one of the two registers (or also via both simultaneously).

To come to the essential point first: You can affect only the DB register with LAD or FBD. If you open a data block with the OPN coil/box (see below), you always open it via the DB register. In LAD and FBD, the instance data block is always opened via the DI register; this happens with the call box when the block is called.

When you load a data word, you must specify which of the two possible open data blocks contains the data word. If the data block has been opened via the DB register, the data word is called DBW; if the data word is in the data block opened via the DI register, it is called DIW. The other data operands are named accordingly (Table 18.1).

Table 18.1 Data Operands

Data operand	Located in a data block opened via the	
	DB register	DI register
Data bit	DBX y.x	DIX y.x
Data byte	DBB y	DIB y
Data word	DBW y	DIW y
Data doubleword	DBD y	DID y

x = Bit address, y = Byte address

18.2.2 Accessing Data Operands

You can use the following methods for accessing data operands:

- ▷ Symbolic addressing with full addressing,
- ▷ Absolute addressing with full addressing and
- ▷ Absolute addressing with partial addressing.

Symbolic access to the data operands in global data blocks requires the minimum system knowledge. For absolute access or for using both data block registers, you must observe the notes below.

Symbolic addressing of data operands

It is recommended that you address data operands symbolically whenever possible. Symbolic addressing

- ▷ makes it easier to read and understand the program (if meaningful terms are used as symbols),
- ▷ reduces write errors made during programming (the Program Editor compares the terms used in the Symbol Table and in the program; “number-switching errors” such as DBB 156 and DBB 165 that can occur when using absolute addresses cannot occur here) and
- ▷ does not require programming knowledge at the machine code level (which data block has the CPU opened currently?).

Symbolic addressing uses fully-addressed access (data block together with data operand), so that the data operand always has a unique address.

You determine the symbolic address of a data operand in two steps:

- 1) Assignment of the data block in the Symbol Table
Data blocks are global data that have unique addresses within a program. In the Symbol Table, you assign a symbol (e.g. Motor1) to the absolute address of the data block (e.g. DB 51).
- 2) Assignment of the data operands in the data block
You define the names of the data operands (and the data type) during programming of the data block. The name applies only in the associated block (it is “block-local”). You can also use the same name in another block for another variable.

Fully-addressed access to data operands

In the case of fully-addressed access, you specify the data block together with the data operand. This method of addressing can be symbolic or absolute:

```
"MOTOR1".ACTVAL
DB 51.DBW 20
```

MOTOR1 is the symbolic address that you have assigned to a data block in the Symbol Table. ACTVAL is the data operand you defined when programming the data block. The symbolic name "MOTOR1".ACTVAL is just as unique a specification of the data operand as the specification DB 51.DBW 20.

Fully-addressed data access is only possible in conjunction with the global data block register (DB register). In the case of fully-addressed data operands, the Program Editor first opens the data block via the DB register and then accesses the data operand.

You can use fully-addressed access with all functions permissible for the data type of the addressed data operand. So with block parameters, for example, you can also specify a fully-addressed data operand as actual parameter.

Absolute addressing of data operands

For absolute addressing of data operands, you must know the addresses at which the Program Editor places the data operands when setting up. You can find out the addresses by outputting them after programming and compiling the data block. You will then see from the address column the absolute address at which the relevant variable begins. This procedure is suitable for all data blocks, both those you use as global data blocks as well as those you use as instance data blocks (for local instances see Chapter 18.2.4, "Special Points in Data Addressing"). In this way, you can also see where the Program Editor stores the block parameters and the static local data for function blocks.

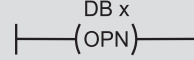
Data operands are addressed byte by byte like, for example, bit memory; the functions used on the data operands are also the same as those used with bit memory.

18.2.3 Opening a Data Block

You use the OPN coil (LAD) or OPN box (FBD) to open a data block via the DB register.

Open data block

LAD representation:



FBD representation:



The OPN coil/box is connected directly to the left power rail and stands alone in a rung. The specified data block is always opened via the DB register. In LAD or FBD, it is not possible to open a data block via the DI register with OPN coil/box. (The DI register uses the call box to open the current instance data block.)

In the case of incremental programming, you will find the OPN coil/box in the Program Elements Catalog under "DB Call".

The open data block must be in the work memory at runtime.

In the networks following the OPN coil/box, you can use partial addressing to access only those data operands that are located in the open data block. If you want to copy from one data block to another, you can, for example go into the temporary local data via an intermediate buffer or (better) use fully-addressed data operands. Please note: A fully-addressed data operand overwrites the DB register with "its" data block.

Example: The value of data word DBW 10 of data block DB 13 is to be transferred to data word DBW 16 of data block DB 14. With the MOVE box, you can only copy partially-addressed data operands within the currently open data block. For copying between data blocks, you require an intermediate buffer, such as a local data word (see Figure 18.7). It is better to use full addressing.

When you open a data block, it remains "valid" until another data block is opened. Under certain circumstances, you may not be able to see this via the Program Editor, for example if a block is changed with the call box (see Chapter 18.2.4, "Special Points in Data Addressing").

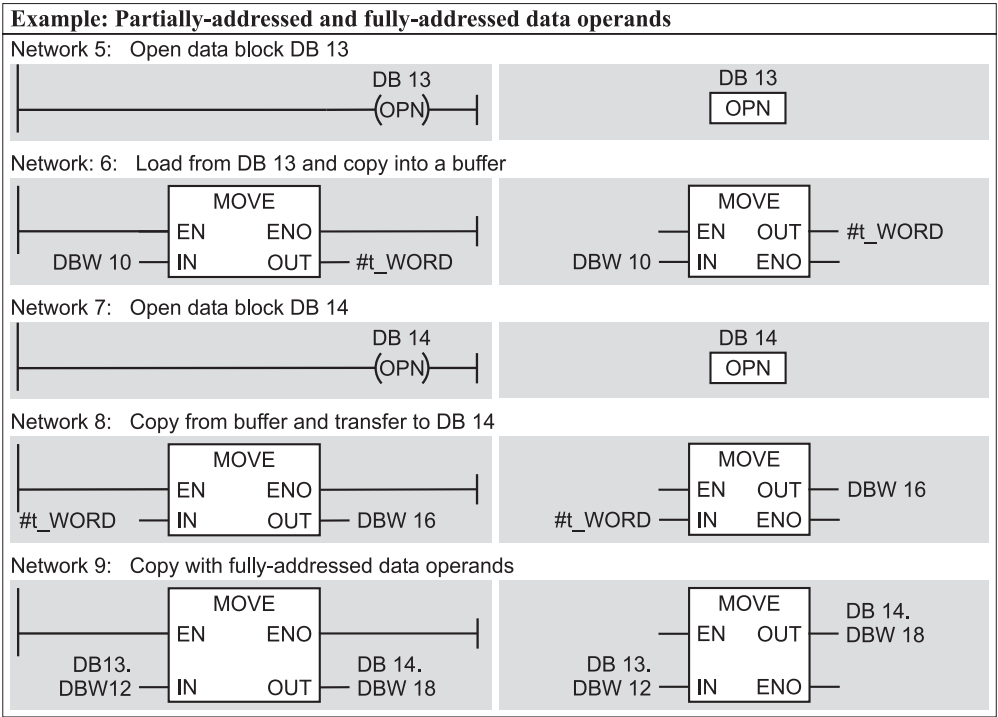


Figure 18.7 Example of Partially- and Fully-Addressed Data Operands

In the case of a block change with the CALL coil/box, the contents of the data block registers are retained. On returning to the calling block, the block change restores the old contents of the registers.

18.2.4 Special Points in Data Addressing

Changes in the contents of the DB registers

With the following functions, the Program Editor generates additional instructions which can alter the contents of the two DB registers:

Full addressing of data operands

Each time data operands are fully addressed, the Program Editor first opens the data block and then accesses the data operand. The DB register is overwritten each time. This also applies to supplying block parameters with fully-addressed data operands.

Accessing block parameters

Access to the following block parameters changes the contents of the DB register: All block parameters of complex data type in the case of functions, and in/out parameters of complex data type in the case of function blocks.

Calling a function block with the call box

Before the actual function block call, the call box stores the number of the current instance data block in the DB register (by swapping the data block registers) and opens the instance data block for the called function block. In this way, the instance data block associated with a called function block is always open. After the actual block call, the call box swaps data block registers again, so that the current instance data block is available again in the calling function block. In this way, the call box changes the contents of the DB register.

DI register in function blocks

In function blocks, the DI register always contains the number of the current instance data block. All access to block parameters or static local data is over the DI register. The address of the local data given in the Declaration Table of the function block applies only when you open the function block with an instance data block, in which case the data operands begin at byte zero.

When you call a function block as local instance, its data are “in the middle” of the calling data block's instance data block. They contain the absolute address of the local data when you output the instance data block in data view format. Each individual variable is then displayed with name and address, including the variables of the local instance called.

When you program a function block that you also want to have available later as local instance, you do not yet know the absolute address of the variable at the time of programming. In this case – just as the Program Editor does for symbolic programming – the contents of address register AR2 are added to the variable address. However, this is possible only in the STL programming language.

Changing the contents of data blocks at a later time

In the Properties window of the offline object container *Blocks* in the “Blocks” register, you can specify whether the absolute address or the symbolic address of the data operand is to have priority for subsequent displays and saves when there is a change in the contents of the data blocks for code blocks that have already been stored.

The default is “Absolute address has priority” (the same as in previous STEP 7 versions). This default means that when there is a change in the declaration, the absolute address is retained in the program while the symbolic address changes accordingly. If the “Symbolic address has priority” default is chosen, the absolute address changes while the symbolic address is retained.

Example: Assuming that data word DBW 10 in data block DB 1 contains the symbolic address of *ActValue*. In the program, you might address this data word with

```
"Data".ActValue    DB1.DBW 10
```

where “Data” is the symbolic address for data block DB 1. If you would now add an additional data word with the symbolic address *MaxCurrent* immediately in front of data word DBW 10, the result upon the next opening (or storing) of the code block will be as follows:

In the case of “*Absolute address has priority*”:

```
"Data".MaxCurrent  DB1.DBW 10
```

In the case of “*Symbolic address has priority*”:

```
"Data".ActValue    DB1.DBW 12
```

As for access to data operands in global data blocks, the same applies as for access to global operands (for instance inputs) for which a symbolic address has been entered in the Symbol Table. You will find detailed information on this subject in Chapter 2.5.5, “Address Priority”.

Note that this “rewiring” does not take place automatically since the blocks which have already been compiled contain the executable MC7 code of the statements with the absolute address. The change is only carried out in the associated block (with a corresponding message) following opening and saving again.

In order to carry out the modification in the complete block folder, select EDIT → CHECK BLOCK CONSISTENCY with the *Blocks* object selected.

18.3 System Functions for Data Blocks

There are three system functions for handling data blocks. Their parameters are described in Table 18.2.

- ▷ SFC 22 CREAT_DB
Create data block in work memory
- ▷ SFC 85 CREA_DB
Create data block in work memory
- ▷ SFC 82 CREA_DBL
Create data block in load memory
- ▷ SFC 23 DEL_DB
Delete data block
- ▷ SFC 24 TEST_DB
Test data block

Table 18.2 SFCs for Handling Data Blocks

SFC	Name	Declaration	Data Type	Assignment, Description
22	LOW_LIMIT	INPUT	WORD	Lowest number of the data block to be created
	UP_LIMIT	INPUT	WORD	Highest number of the data block to be created
	COUNT	INPUT	WORD	Length of the data block in bytes (even number)
	RET_VAL	RETURN	INT	Error information
	DB_NUMBER	OUTPUT	WORD	Number of the created data block
85	LOW_LIMIT	INPUT	WORD	Lowest number of the data block to be created
	UP_LIMIT	INPUT	WORD	Highest number of the data block to be created
	COUNT	INPUT	WORD	Length of the data block in bytes (even number)
	ATTRIB	INPUT	BYTE	Block attribute: B#16#00 Retain B#16#04 Non_Retain
	RET_VAL	RETURN	INT	Error information
	DB_NUMBER	OUTPUT	WORD	Number of the created data block
82	REQ	INPUT	BOOL	Trigger for generation with signal state “1”
	LOW_LIMIT	INPUT	WORD	Lowest number of the data block to be created
	UP_LIMIT	INPUT	WORD	Highest number of the data block to be created
	COUNT	INPUT	WORD	Length of the data block in bytes (even number)
	ATTRIB	INPUT	BYTE	Attributes of created data block
	SRCBLK	INPUT	ANY	Data area in work memory with which the created data block is initialized
	RET_VAL	RETURN	INT	Error information
	BUSY	OUTPUT	BOOL	If TRUE, creation has not yet been finished
23	DB_NUM	OUTPUT	WORD	Number of the created data block
	DB_NUMBER	INPUT	WORD	Number of the data block to be deleted
	RET_VAL	RETURN	INT	Error information
24	DB_NUMBER	INPUT	WORD	Number of the data block to be tested
	RET_VAL	RETURN	INT	Error information
	DB_LENGTH	OUTPUT	WORD	Length of the data block (in bytes)
	WRITE_PROT	OUTPUT	BOOL	TRUE = write-protected

18.3.1 Creating a Data Block in Work Memory

System functions SFC 22 CREAT_DB and SFC 85 CREA_DB create a data block in the work memory. As the data block number, the system function takes the lowest free number in the number band given by the input parameters LOW_LIMIT and UP_LIMIT. The numbers specified at these parameters are included in the number band. If both values are the same, the data block is created with this number. The number of a data block already present in the user program cannot be assigned again, not

even if the data block is only present in the load memory.

The output parameter DB_NUMBER supplies the number of the data block actually created. With the input parameter COUNT, you specify the length of the data block to be created. The length corresponds to the number of data bytes and must be an even number.

Creating the data block is not the same as calling it. The current data block is still valid. A data block created with this system function contains random data. For meaningful use, data

must first be written to a data block created in this way before the data can be read.

The data blocks created with the SFCs 22 CREAT_DB and 85 CREA_DB are only present in work memory. If a CPU differentiates between retentive and non-retentive work memory, the SFC 22 CREAT_DB creates a retentive data block, and the SFC 85 CREA_DB creates a data block as specified by the ATTRIB parameter. “Retentive” data block means that its contents are retained in the event of a warm restart/hot restart (see Chapter 22.2.4, “Retentivity”).

The system function SFC 85 CREA_DB replaces the SFC 22 CREAT_DB.

A data block created by the SFCs 22 CREAT_DB and SFC 85 CREA_DB does not change the checksum of the user program, not even if it is written or deleted again. If a data block created by an SFC is imported into the offline data management, this has an influence on the checksum.

In the event of an error, a data block is not created, the parameter DB_NUMBER parameter is assigned zero, and an error number is output by RET_VAL.

18.3.2 Creating a Data Block in Load Memory

System function SFC 82 CREA_DBL creates a data block in the load memory. As the data block number, the system function takes the lowest free number in the number band given by the input parameters LOW_LIMIT and UP_LIMIT. The numbers specified at these parameters are included in the number band. If both values are the same, the data block is created with this number. The number of a data block already present in the user program cannot be assigned again, not even if the data block is only present in the work memory.

The output parameter DB_NUM supplies the number of the data block actually created. With the input parameter COUNT, you specify the length of the data block to be created. The length corresponds to the number of data bytes and must be an even number.

The data area assigned as default to the created data block is specified at input parameter SRCBLK. Here you can specify a complete data block, e.g. DB 160 or “Archive 1”, a variable from a data block, or an absolute addressed data area as an ANY pointer, e.g. P#DB160.DBX16.0 BYTE 64. The source must be a data area in the work memory.

If the source area is smaller than the target area, the source area is written completely into the target area. The remaining bytes of the target area are filled with zeros. If the source area is larger than the target area, the target area is written completely; the remaining bytes of the source area are ignored.

You can assign the following attributes to the created data block using the ATTRIB parameter:

- ▷ Bit 0 = “1”
The data block has the *Unlinked* attribute. Following transfer to the offline data management and reloading into the CPU, the data block is still only present in the load memory.
- ▷ Bit 1 = “1”
The data block has the *DB is write-protected in the PLC* attribute. You can only read the values of this data block.
- ▷ Bit 2 = “1”
The data block has the *Non_Retain* attribute.

The other bits are not used at the moment. You can find further information on the block attributes in Chapter 3.2.3, “Block Properties”

System function SFC 82 CREA_DBL operates in asynchronous mode: you trigger the creation process with signal state “1” at input parameter REQ. You may only access the read and written data areas again when the output parameter BUSY has returned to the signal state “0”.

Creating does not call the associated data block. The current data block is still the valid one.

A data block is not created in the event of an error, the output parameters are assigned undefined values, and an error message is output by the function value.

18.3.3 Deleting a Data Block

System function SFC 23 DEL_DB deletes the data block in RAM (work and load memory) whose number is specified in the input parameter DB_NUMBER. The data block must not be open at the time, as otherwise the CPU will go to STOP.

Data blocks created with the keyword *Unlinked* and data blocks on an Flash EPROM memory card cannot be deleted with system function SFC 23.

In the event of an error, the data block is not deleted and an error number is returned as function value.

18.3.4 Testing a Data Block

System function SFC 24 TEST_DB provides information on a data block whose number you specify at input parameter DB_NUMBER. Output parameter DB_LENGTH contains the number of existing bytes, and output parameter WRITE_PROT indicates whether the data block is write-protected.

If the tested data block is only in load memory, this is indicated as an error by RET_VAL; the DB_LENGTH and WRITE_PROT parameters nevertheless have the correct assignments.

If the specified data block is not present in the CPU's work memory, RET_VAL is returned with W#16#80B1.

19 Block Parameters

This chapter describes how to use block parameters. You will learn

- ▷ how to declare block parameters,
- ▷ how to work with block parameters,
- ▷ how to initialize block parameters and
- ▷ how to 'forward' block parameters.

Block parameters represent the transfer interface between the calling and the called block. All functions and function blocks can be provided with block parameters.

19.1 Block Parameters in General

19.1.1 Defining the Block Parameters

Block parameters make it possible to parameterize the processing instruction in a block, the

block function. Example: You want to write a block as an adder that you can use in your program several times with different variables. You transfer the variables as block parameters; in our example, three input parameters and one output parameter (Figure 19.1). Since the adder need not store any values internally, a function is suitable as the block type.

You define a block parameter as an input parameter if you only check or load its value in the block program. If you only describe a block parameter (assign, set, reset, transfer), you use an output parameter. You must always use an in/out parameter if a block parameter is to be both checked and overwritten. The Program Editor does not check the use of the block parameters.

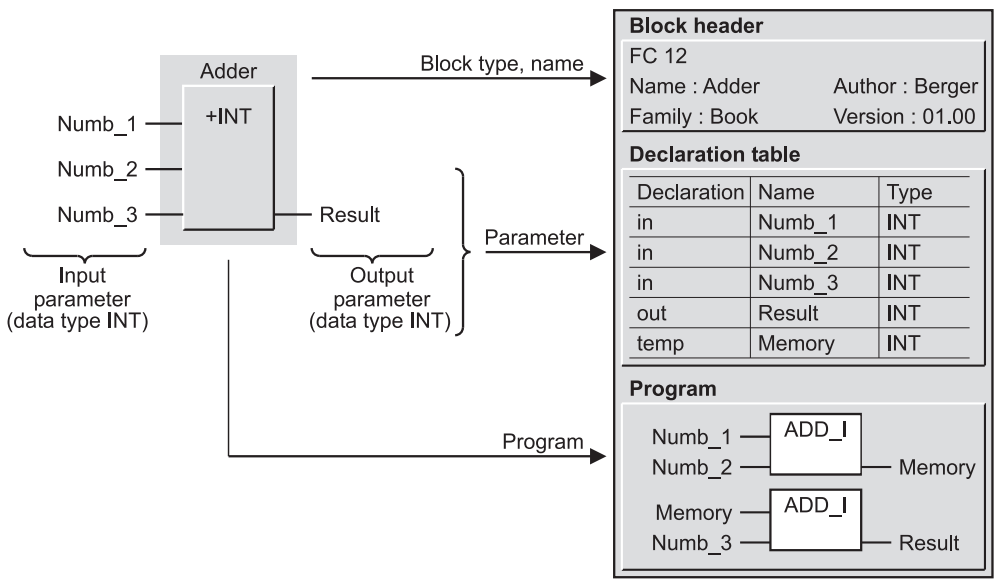


Figure 19.1 Example of Block Parameters

19.1.2 Processing the Block Parameters

In the adder program, the names of the block parameters stand as place holders for the latest actual variables. You use the block parameters in the same way as symbolically addressed variables; in the program, they are called “formal parameters”.

You can call the “Adder” function several times in your program. With each call, you transfer other values to the adder in the block parameters (Figure 19.2). The values can be constants, operands or variables; they are called “actual parameters”.

At runtime, the CPU replaces the formal parameters with the actual parameters. The first call in the example adds the contents of memory words MW 30, MW 32 and MW 34 and stores the result in memory word MW 40. The same block with the actual parameters of the second call adds data words DBW 30, DBW 32 and DBW 34 of data block DB 13 and stores the result in data word DBW 40 of data block DB 14.

19.1.3 Declaration of the Block Parameters

You define the block parameters in the declaration section of the block when you program the block. Figure 19.3 shows the declaration tables of a function FC and a function block FB. In addition to the block parameters (IN, OUT, IN_OUT), you also declare the temporary local data (TEMP), the function value (RETURN) for functions FC, and the static local data (STAT) for function blocks. Only the temporary local data (TEMP) exists for organization blocks OB which are neither called in the user program nor possess an instance data block.

Default values are optional and are only possible for function blocks if a block parameter is stored as a value. This applies to all block parameters of elementary data type and to input and output parameters of complex data type. A parameter comment can also be given.

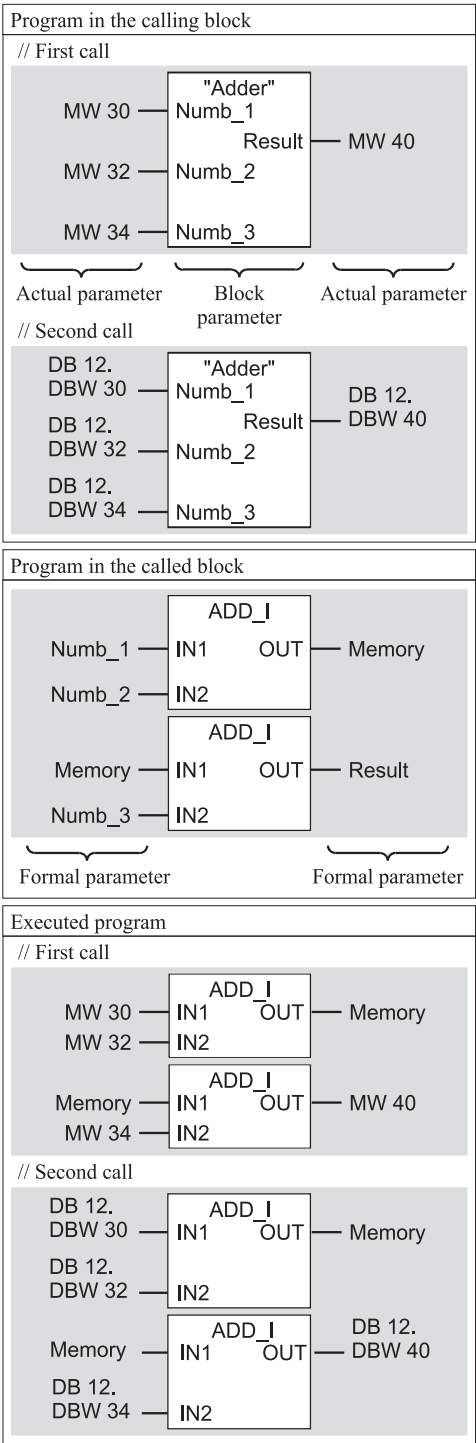


Figure 19.2 Block Call with Block Parameters

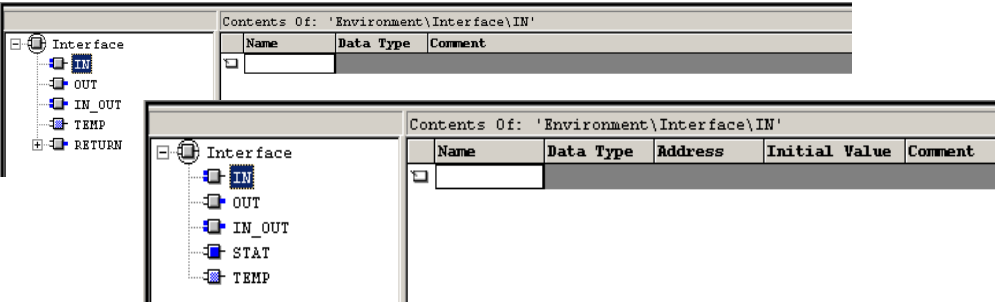


Figure 19.3 Empty Declaration Tables for Functions FC and Function Blocks FB

The *block parameter name* can be up to 24 characters in length. It must consist only of letters (without national characters such as the German Umlaut), digits, and the underscore. No distinction is made between upper and lower case. The name must not be a keyword.

For the *data type* of a block parameter, all elementary, complex and user-defined data types are permissible as well as the parameter types (see Chapter 3.5, “Variables, Constants and Data Types”).

STEP 7 stores the names of the block parameters in the non-executable section of the blocks on the programming device's storage medium. The work memory of the CPU (in the compiled block) contains only the declaration types and the data types. For this reason, program changes made to blocks online in the CPU must always be updated on the programming device's data medium in order to retain the original names. If the update is not made, or if blocks are transferred from the CPU to the programming device, the non-executable-block sections are overwritten or deleted. The Program Editor then generates replacement symbols for display or printout.

19.1.4 Declaration of the Function Value

In functions, the function value is a specially treated output parameter. It has the name RET_VAL (or ret_val) and is defined as the first output parameter. You declare the function value

by specifying the data type and possibly a comment in the declaration table under RETURN and the name RET_VAL.

As data type of the function value, all elementary data types as well as the data types DATE_AND_TIME, STRING, POINTER, ANY and user-defined data types UDT are permitted. The data types ARRAY and STRUCT are not allowed.

As the first output parameter, the function value has no special role to play in the LAD or FBD programming language. It only gains significance in the SCL programming language, where you can use the block type FUNCTION as a “genuine” function. Here, a function FC can stand in place of an operand in a printout; the function value then represents the value of this function.

19.1.5 Initializing Block Parameters

When calling a block with block parameters, you initialize the block parameters with actual parameters. These can be constants, operands with absolute addresses, fully-addressed data operands or symbolically addressed variables. The actual parameter must be of the same data type as the block parameter.

You must initialize all of a function's block parameters at every call. In the case of function blocks, initialization of individual block parameters or all block parameters is optional.

19.2 Formal Parameters

In this chapter, you will learn how to access the block parameters within a block. Table 19.1 shows that it is possible to access block parameters of elementary data type, components of an array or structure, and timers and counters without restriction. Chapter 19.4, ““Forwarding” Block Parameters”.

Access to parameters of complex data type and of parameter types POINTER and ANY is not supported by LAD or FBD. However, you can initialize acquired blocks or system blocks that have such parameters with the relevant variable. You can find examples of this in the libraries “LAD_Book” and “FBD_Book” under the program “Data Types” that you can download from the publisher's Website (see page 8).

Block parameters with data type BOOL

Block parameters of data type BOOL can be individual binary variables or binary compo-

nents of arrays and structures. You can check input parameters and in/out parameters with contacts or with binary box inputs, and you can influence output parameters and in/out parameters with binary box outputs. After the CPU has used the actual parameter specified as block parameter, it processes the functions as shown in the relevant chapters.

Block parameters of digital data type

Block parameters of digital data type occupy 8, 16 or 32 bits (all elementary data types except BOOL). They can be individual digital variables or digital components of arrays and structures. You apply input and in/out parameters to digital box inputs, and you write output and in/out parameters to digital box outputs.

You exchange values between block parameters of different types or different sizes with the MOVE box as described in Chapter 6, “Move Functions”.

Table 19.1 Accessing Block Parameters (General)

Data Types	Permitted for			Access in Block Possible
	IN	I_O	OUT	
Elementary data types				
BOOL	x	x	x	yes
BYTE, WORD, DWORD, CHAR, INT, DINT, REAL, S5TIME, TIME, TOD, DATE	x	x	x	yes
Complex data types				
DT, STRING	x	x	x	no
ARRAY, STRUCT				
Individual binary components	x	x	x	yes
Individual digital components	x	x	x	yes
Complete variables	x	x	x	no
Parameter types				
TIMER	x	-	-	yes
COUNTER	x	-	-	yes
BLOCK_FC, BLOCK_FB	x	-	-	yes
BLOCK_DB	x	-	-	yes
BLOCK_SDB	x	-	-	no
POINTER, ANY	x	x	x ¹⁾	no

¹⁾ FC functions only

Block parameters of data type DT and STRING

Direct access to block parameters of type DT and STRING is not possible. In function blocks, you can “pass” input and output parameters of data type DT and STRING to parameters of called block.

Block parameters of data type ARRAY and STRUCT

Direct access to block parameters of type ARRAY and STRUCT is possible on a component-by-component basis, that is, you can access individual binary or digital components with the relevant operations.

Access to the complete variable (entire array or entire structure) is not possible and neither is access to individual components of complex or user-defined data type. In function blocks, you can “pass” input and output parameters of data type ARRAY and STRUCT to parameters of called blocks.

Block parameters of user-defined data type

You handle block parameters of user-defined data type in the same way as block parameters of data type STRUCT.

Direct access to block parameters of data type UDT is possible on a component-by-component basis, that is, you can access individual binary or digital components with the relevant operations.

Access to the complete variable is not possible and neither is access to individual components of complex or user-defined data type. In function blocks, you can “pass on” input and output parameters of data type UDT to parameters of called blocks.

Block parameters of data type TIMER

You can use block parameters of data type TIMER with all functions as described in Chapter 7, “Timers”. When a timer is started, the time value can also be a block parameter of data type S5TIME.

Block parameters of data type COUNTER

You can use block parameters of data type COUNTER with all functions as described in Chapter 8, “Counters”. When setting a counter, the counter value can also be a block parameter of data type WORD.

Block parameters of data type BLOCK_DB

You can transfer a data block via a block parameter of data type BLOCK_DB. Call this data block with the OPN coil/box by labeling the OPN coil/box with the formal parameters. When opening a data block via a block parameter, the CPU always uses the global data block register (DB register).

Block parameters of data type BLOCK_FC

You can transfer an FC function via a block parameter of data type BLOCK_FC. Call this function with the CALL coil/box. You can use the CALL coil/box with a formal parameter and with or without any preceding logic operation if you are currently programming a function block. If you use the CALL coil/box with formal parameters in a function, a preceding logic operation is not permissible (absolute call only).

An FC function transferred via a block parameter must not have any block parameters.

Block parameters of data type BLOCK_FB

You can transfer an FB function block via a block parameter of data type BLOCK_FB. Direct access to block parameters of data type BLOCK_FB is not possible with LAD or FBD functions.

An FB function block transferred via a block parameter must not have any block parameters.

Block parameters of data type POINTER and ANY

Direct access to block parameters of data type POINTER and ANY is not possible with LAD or FBD functions.

19.3 Actual Parameters

When you call a block, you initialize its block parameters with constants, operands or variables with which it is to operate. These are the actual parameters. If you call the block often in your program, you usually use different actual parameters each time.

An actual parameter must agree in data type with the block parameter: You can only apply a binary actual parameter (for example, a memory bit) to a block parameter of type BOOL; you can only initialize a block parameter of type ARRAY with an identically dimensioned array variable. Table 19.2 gives an overview of which operands you can use as actual parameters with which data type.

When calling functions, you must initialize all block parameters with actual parameters.

When calling function blocks, it is not a mandatory requirement that you initialize block parameters with actual parameters. If you make no specification at a block parameter, the (old) values stored in the instance data block are used as the actual parameters. These can be, for example, the default values or the values of actual parameters from an earlier call. In/out parameters of complex data type cannot be assigned default values, and neither can parameter types. You must provide these block parameters with actual parameters at least at the first call.

You can also use direct access to access the block parameters for the function block. Since they are located in a data block, you can handle the block parameters like data operands. Example: A function block with the instance data block *“Station_1”* controls a binary output parameter with the name *“Up”*. Following processing in the function block (after its call), you can check the parameter under the symbolic address *“Station_1”.Up*, without having initialized the output parameter.

Initializing block parameters with elementary data types

The actual parameters listed in Table 19.3 are permissible as actual parameters of elementary data type.

You can assign either absolute or symbolic addresses to input, output and bit memory operands. Input operands should be used only for input parameters and output operands for output parameters (however, this is not mandatory). Bit memory operands are suitable for all declaration types. You must apply peripheral inputs only to input parameters and peripheral outputs only to output parameters.

You use the Boolean constants TRUE and FALSE in LAD and FBD to set defaults in the declaration of block parameters, static local data or data operands. Initializing block parameters with these constants is not permitted. You also cannot create these constants at a contact (LAD) or at a function input (FBD). If you want

Table 19.2 Initialization with Actual Parameters

Data Type of the Block Parameter	Permissible Actual Parameters
Elementary data type	<ul style="list-style-type: none"> ▷ Simple operands, fully-addressed data operands, constants ▷ Components of arrays or structures of elementary data type ▷ Block parameters of the calling block ▷ Components of block parameters of the calling block of elementary data type
Complex data type	<ul style="list-style-type: none"> ▷ Variables or block parameters of the calling block
TIMER, COUNTER and BLOCK_xx	<ul style="list-style-type: none"> ▷ Timers, counters and blocks
POINTER	<ul style="list-style-type: none"> ▷ Simple operands, fully-addressed data operands ▷ Range pointer or DB pointer
ANY	<ul style="list-style-type: none"> ▷ Variables of any data type ▷ ANY pointer

Table 19.3 Actual Parameters of Elementary Data Type

Operands	Permissible with			Binary operand or symbolic name	Digital operand or symbolic name
	IN	I_O	OUT		
Inputs (process image)	x	x	x	I y.x	EB y, EW y, ED y
Outputs (process image)	x	x	x	Q y.x	AB y, AW y, AD y
Bit memory	x	x	x	M y.x	MB y, MW y, MD y
Peripheral inputs	x	-	-	-	PEB y, PEW y, PED y
Peripheral outputs	-	-	x	-	PAB y, PAW y, PAD y
Global data					
Partial addressing	x	x	x	-	DBB y, DBW y, DBD y
Full addressing	x	x	x	DB z.DBX y.x	DB z.DBB y, etc.
Temporary local data	x	x	x	L y.x	LB y, LW y, LD y
Static local data	x	x	x	DIX y.x	DIB y, DIW y, DID y
Constants	x	-	-	-	All digital constants
Components of ARRAY or STRUCT	x	x	x	Complete component name	Complete component name

x = Bit number, y = Byte address, z = Data block number

to initialize block parameters with TRUE or FALSE, you require a variable that is permanently assigned signal state “1” or “0” and that you use instead of the Boolean constants. Bit memories are obvious candidates for variables in this case. The following example shows you how to set a bit memory permanently to signal state “1” or “0” at startup, see Figure 19.4.

Part-addressed data bits are not permissible as actual parameters in LAD and FBD. The reason for this is that the Program Editor transfers the signal state of every binary actual operand to a temporary local data bit prior to the actual block call. For this reason, it is possible in LAD and FBD to initialize a binary block input with a logic operation. During copying before the block call, the assignment of the data block to the data bit is lost. Even if you open the “correct” data block in the program of the called block prior to accessing the block parameter occupied by a partially addressed data bit, only the previously created copy of the signal state will be scanned.

The same applies to binary in/out parameters and output parameters: here too, assignment with part-addressed data bits is not permissible (the Program Editor will not reject such an assignment but in most cases it will result in errored functions).

When using part-addressed data operands at digital parameters, you must note that when accessing the block parameters (in the called block), the currently opened data block is also the “correct” data block. Since the Editor may in certain circumstances, such as during a block call or if the block parameters have previously been accessed, not visibly change the data block, the use of partially addressed digital data operands is not recommended. Use only fully addressed data operands for this reason.

Temporary local data are usually symbolically addressed. They are located in the L stack of the calling block and are declared in the calling block.

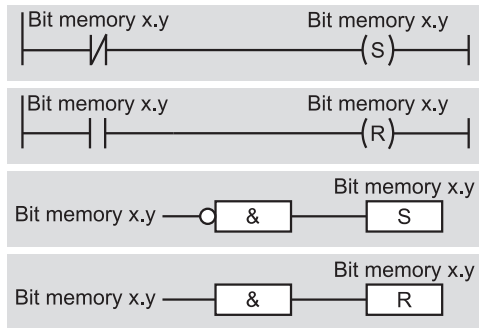


Figure 19.4 Initialization with Fixed Signal State

If the calling block is a function block, you can also use its static local data as actual parameters (see Chapter 19.4, ““Forwarding” Block Parameters”). Static data are usually symbolically addressed; if you would like to assign absolute addresses via DI operands, please note the information in Chapter 18.2.4, “Special Points in Data Addressing”.

With a block parameter of type BOOL, you can apply the constant TRUE (signal state “1”) or FALSE (signal state “0”), and with block parameters of digital data type, you can apply all constants corresponding to the data type. Initialization with constants is permissible only for input parameters.

You can also initialize a block parameter of elementary data type with components of arrays and structures if the component is of the same data type as the block parameter.

Initializing block parameters of complex data type

Every block parameter can be of the complex type. Variables of the same type can be used as actual operands.

For initializing block parameters of type DT or STRING, individual variables or components of arrays or structures of the same data type are permissible.

STRING variables can be of variable length. If you have specified the STRING length when declaring an input or output parameter for a function block, the Editor reserves the specified space in the instance data block; if you have not specified any length, 256 bytes are reserved for the STRING variable. The maximum length of the STRING variable, which you specify in the declaration, must be the same for the actual parameter and the formal parameter. Exception: With FC functions, you specify either no length or the standard length of 254 bytes when declaring a STRING variable; here, you can use STRING variables of all lengths as actual parameters.

For initializing block parameters of type ARRAY or STRUCT, variables with exactly the same structure as the block parameters are permissible.

Initializing block parameters of user-defined data type

For complex or extensive data structures, the use of user-defined data types (UDTs) is recommended. First, you define the UDT and then you use it, for example, to generate the variable in the data block or to declare the block parameter. You can then use the variable when initializing the block parameter. Here, too, the actual parameter (the variable) must be of the same data type (the same UDT) as the block parameter.

A complete data block with the same UDT type as the block parameter is not permissible as actual parameter.

Initializing block parameters of type TIMER, COUNTER and BLOCK_xx

You initialize a block parameter of type TIMER with a timer and a block parameter of type COUNTER with a counter. You can apply only blocks without their own parameters to block parameters of parameter types BLOCK_FC and BLOCK_FB. You initialize BLOCK_DBs with a data block.

Block parameters of types TIMER, COUNTER and BLOCK_xx must be input parameters.

Initializing block parameters of type POINTER

Pointers (constants) and operands are permissible for block parameters of parameter type POINTER. These pointers are either range pointers (32-bit pointers) or DB pointers (48-bit pointers). The operands are of elementary data type and can also be fully-addressed data operands.

Output parameters of type POINTER are not permitted for function blocks.

Initializing block parameters of type ANY

Variables of all data types are permissible for block parameters of parameter type ANY. The programming within the called block determines which variables (operands or data types) must be applied to the block parameters, or which variables are feasible. You can also specify a constant in the format of the ANY pointer

Table 19.4 Permissible Combinations when Forwarding Block Parameters

Calling → called Declaration type	FC calls FC			FB calls FC			FC calls FB			FB calls FB		
	E	C	P	E	C	P	E	C	P	E	C	P
Input → Input	x	-	-	x	x	-	x	-	x	x	x	x
Output → Output	x	-	-	x	x	-	x	-	-	x	x	-
In/out → Input	x	-	-	x	-	-	x	-	-	x	-	-
In/out → Output	x	-	-	x	-	-	x	-	-	x	-	-
In/out → In/out	x	-	-	x	-	-	x	-	-	x	-	-

E = Elementary data types

C = Complex data types

P = Parameter types TIMER, COUNTER and BLOCK_xx

“P#[data block.]Operand Datatype Number” and thus define an absolute-addressed area.

Output parameters of type ANY are not permissible for function blocks.

Block parameters of parameter types TIMER, COUNTER and BLOCK_xx can only be passed on from one input parameter to another if the calling block is a function block. These statements are represented in Table 19.4.

19.4 “Forwarding” Block Parameters

“Forwarding” block parameters is a special form of access and of initializing block parameters. The block parameters of the calling block are “forwarded” to the parameters of the called block. The formal parameter of the calling block then becomes the actual parameter of the called block.

In general, it is also the case here that the actual parameter must be of the same type as the formal parameter (that is, the relevant block parameters must agree in their data types). In addition, you can apply an input parameter of the calling blocks only to an input parameter of the called block, and similarly, an output parameter to an output parameter. You can apply an in/out parameter of the calling block to all declaration types of the called block.

There are restrictions with regard to data types caused by the variations in the storage of block parameter for functions and those for function blocks. Block parameters of elementary data type can be passed on without restriction in accordance with the information in the previous paragraph. Input and output parameters of complex type can only be forwarded if the calling block is a function block.

19.5 Examples

19.5.1 Conveyor Belt Example

The example shows the transfer of signal states via block parameters. For this purpose, we use the function of a conveyor belt control system described in Chapter 5, “Memory Functions”. The conveyor belt control system is to be programmed in a function block, and all inputs and outputs are to be routed through block parameters so that the function block can be called repeatedly (for several conveyor belts). Figure 19.5 shows the input and output parameters for the function block as well as the static and temporary local data used.

Distributing the parameters is quite simple in this case: All binary operands that were inputs have become input parameters, all outputs have become output parameters, and all memory bits have become static local data. You will also have noticed that the names have been altered slightly because only letters, digits, and the underscore are permitted for block-local variables.

The function block “Conveyor_Belt” is to control two conveyor belts. For this purpose, it will be called twice; the first time with the inputs and outputs of conveyor belt 1 and the second

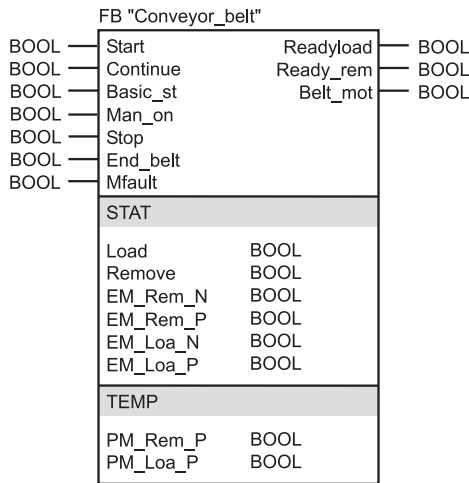


Figure 19.5
Function Block for the Conveyor Belt Example

time with those of conveyor belt 2. For each call, the function block requires an instance data block where it stores the data for the obtaining conveyor belt. The data block for conveyor belt 1 is to be called “BeltData1” and the data block for conveyor belt 2 is to be called “BeltData2”.

You can find the executed programming example in the library “LAD_Book” or “FBD_Book” under the program “Conveyor Example” that you can download from the publisher's Website (see page 8). It shows the programming of function block FB 21 with input parameters, output parameters and static local data. You can use any data blocks as instance blocks; in the example, DB 21 is used for “Belt_data1” and DB 22 for “Belt_data2”. In the Symbol Table, these data blocks have the data type of the function block (FB 21 in the example, if “Conveyor_belt” is the symbol for FB 21).

When you call the function block, you can use the inputs and outputs from the Symbol Table as actual parameters. In those cases where these global symbols contain special characters, you must place these symbols in quotation marks in the program. The Symbol Table is designed for all three examples in this chapter (Table 19.6 at the end of this chapter).

19.5.2 Parts Counter Example

The example demonstrates the handling of block parameters of elementary data type. The “Parts Counter” example from Chapter 8, “Counters”, is the basis of the function. The same function is implemented here as a function block, with all global variables declared either as block parameters or as static local data. The timers and counters are controlled here via individual elements.

Timers and counters are transferred via block parameters of type TIMER and COUNTER. These block parameters must be input parameters. The initial values of the counter (*Quantity*) and the timer (*Dura1* and *Dura2*) can also be transferred as block parameters; the data types of the block parameters correspond here to those of the actual parameters.

The edge memory bits are stored in the static local data and the pulse memory bits are stored in the temporary local data.

You can find the sample program in the library “LAD_Book” or “FBD_Book” under the program “Conveyor Example”, that you can download from the publisher's Website (see page 8). It contains function block FB 22 “Parts_counter” and the associated instance data block

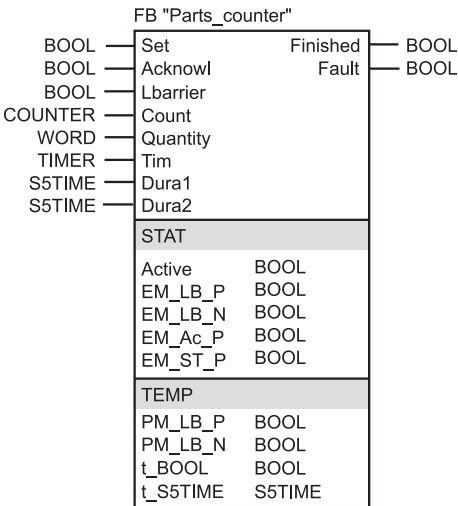


Figure 19.6
Function Block for the Parts Counter Example

“CountDat”. You can use the inputs, outputs, timer and counter from the Symbol Table (Table 19.6 in previous example) as actual parameters when calling the function block.

19.5.3 Feed Example

The same functions described in the two previous examples can also be called as local instances. In our example, this means that we program a function block called “Feed” that is to control four conveyor belts and count the conveyed parts. In this function block, FB “Conveyor Belt” is called four times and FB “Parts_counter” is called once. The call does not take place in each case with its own instance data block, but the called FBs are to store their data in the instance data block of function block “Feed”.

Figure 19.7 shows how the individual conveyor belt controls are interconnected (FB “Parts_counter” is not represented here). The start signal is connected to the *Start* input of the controller for belt 1, the *ready_rem* output is connected to the *Start* input of belt 2, etc. Finally, the *ready_rem* output of belt 4 is connected to the *Remove* output of “Feed”. The same signal sequence leads in the reverse direction from *Remove* over *Continue* and *Readyload* to *Load*.

Belt_mot, *Lbarr* and */Mfault* are separate conveyor belt signals; *Reset*, *Startup* and *Stop* con-

trol all conveyor belts via *Basic_st*, *Man_on* and *Stop*.

The following program for function block “Feed” is designed in the same way. The input and output parameters of the function block can be taken from Figure 19.7. In addition, the digital values for the parts counter *Quantity*, *Dura1* and *Dura2* are designed as input parameters here. We declare the data of the individual conveyor belt controls and the data of the parts counter in the static local data in exactly the same way as for a user-defined data type, i.e. with name and data type. The variable “Belt1” is to receive the data structure of function block “Conveyor Belt”, as is variable “Belt2”, etc.; the variable “Check” receives the data structure of the function block “Parts_counter”.

The program in the function block starts with the initialization of the signals common to all conveyor belts. Here, we make use of the fact that the block parameters of the function blocks called as local instances are static local data in the current block and can be treated as such. The block parameter *Man_on* in the current function block controls the *Man_on* input parameters of all four conveyor belt controls with a single assignment. We proceed in the same way with the signals *Stop* and *Reset*. And now the conveyor belt controls are initialized with the common signals. (You can, of course, also initialize these input parameters when the function block is called.)

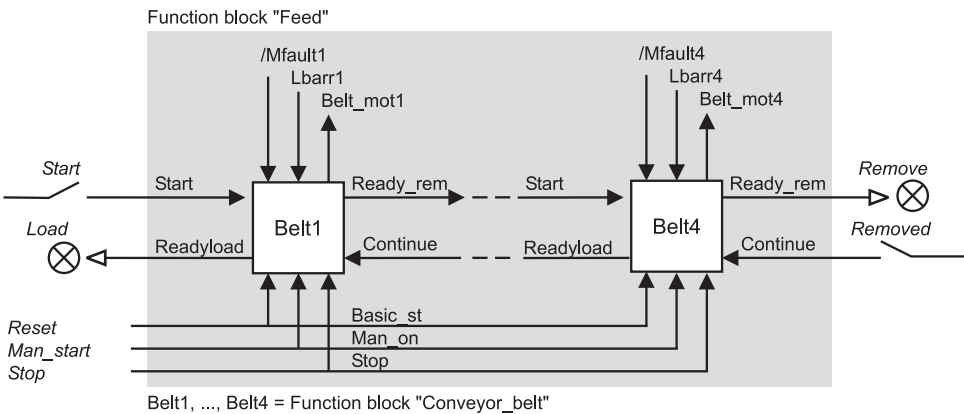


Figure 19.7 Feed Programming Example

Table 19.5 Declaration Section of FB “Feed”

Declara- tion	Name	Data Type	Ad- dress	Initial Value	Comment
IN	Start	BOOL	0.0	FALSE	Start conveyor belts
	Removed	BOOL	0.1	FALSE	Parts have been removed from belt
	Man_start	BOOL	0.2	FALSE	Start conveyor belts manually
	Stop	BOOL	0.3	FALSE	Stop conveyor belts
	Reset	BOOL	0.4	FALSE	Set control to the basic setting
	Count	COUNTER	2.0		Counter for the parts
	Quantity	WORD	4.0	W#16#200	Number of parts
	Tim	TIMER	6.0		Timer for the monitor
	Dura1	S5TIME	8.0	S5T#5S	Monitoring time for parts
	Dura2	S5TIME	10.0	S5T#10S	Monitoring time for gap
OUT	Load	BOOL	12.0	FALSE	Load new parts onto belt
	Remove	BOOL	12.1	FALSE	Remove parts from belt
IN_OUT					
STAT	Belt1	Conveyor_belt	14.0		Control for belt 1
	Belt2	Conveyor_belt	20.0		Control for belt 2
	Belt3	Conveyor_belt	26.0		Control for belt 3
	Belt4	Conveyor_belt	32.0		Control for belt 4
	Check	Parts_counter	38.0		Control for counting and monitoring
TEMP					

The subsequent calls of the function blocks for conveyor belt control contain only the block parameters for the individual signals for each conveyor belt and the connection to the block parameters of “Feed”. The individual signals are the light barriers, the commands for the belt motor and the motor faults. (We make use here of the fact that when a function block is called, not all block parameters have to be initialized.) We program the connections between the individual belt controllers using assignments.

The FB “Parts_counter” is called as a local instance even if it has no closer connection with the signals of the conveyor belt controls. The instance data block for “Feed” stores the FB data.

The input parameters *Quantity*, *Dura1* and *Dura2* for “Feed” need to be set only once. This can be done using default values (as in the example) or in the restart routine in OB 100 (for example, through direct assignment if these three parameters are treated as global data).

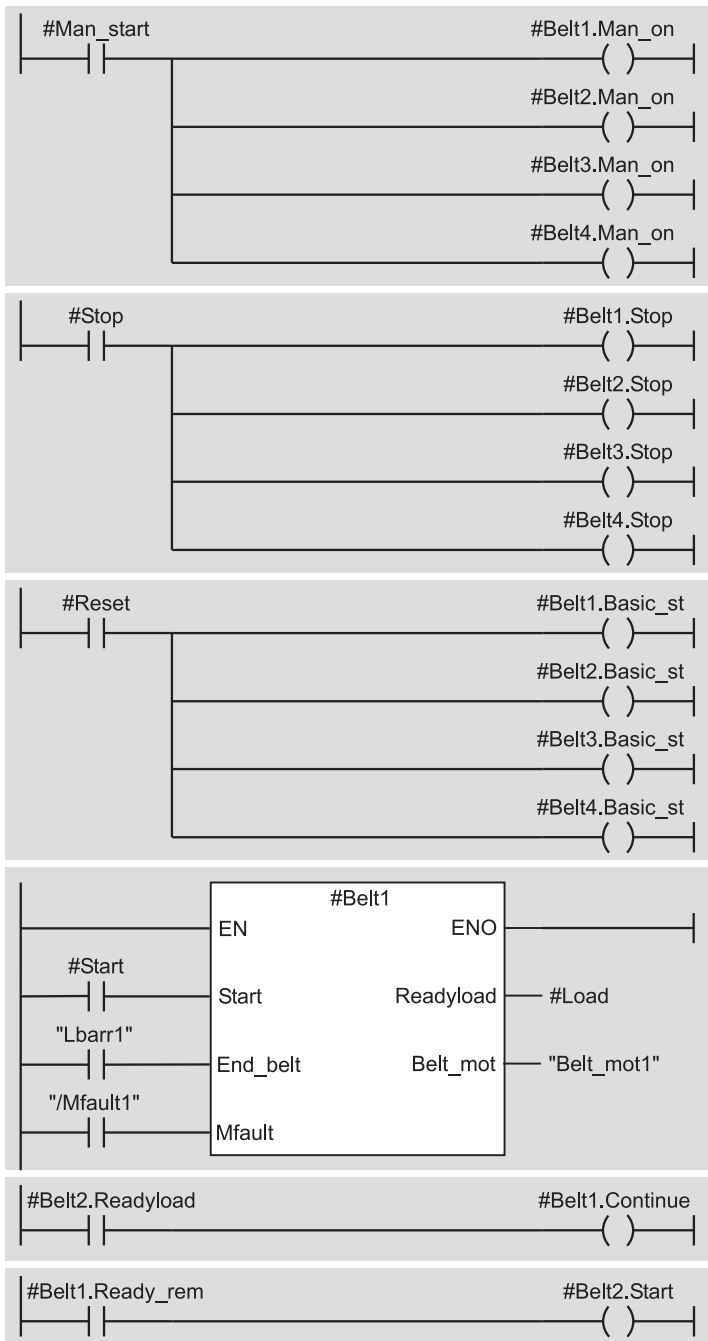
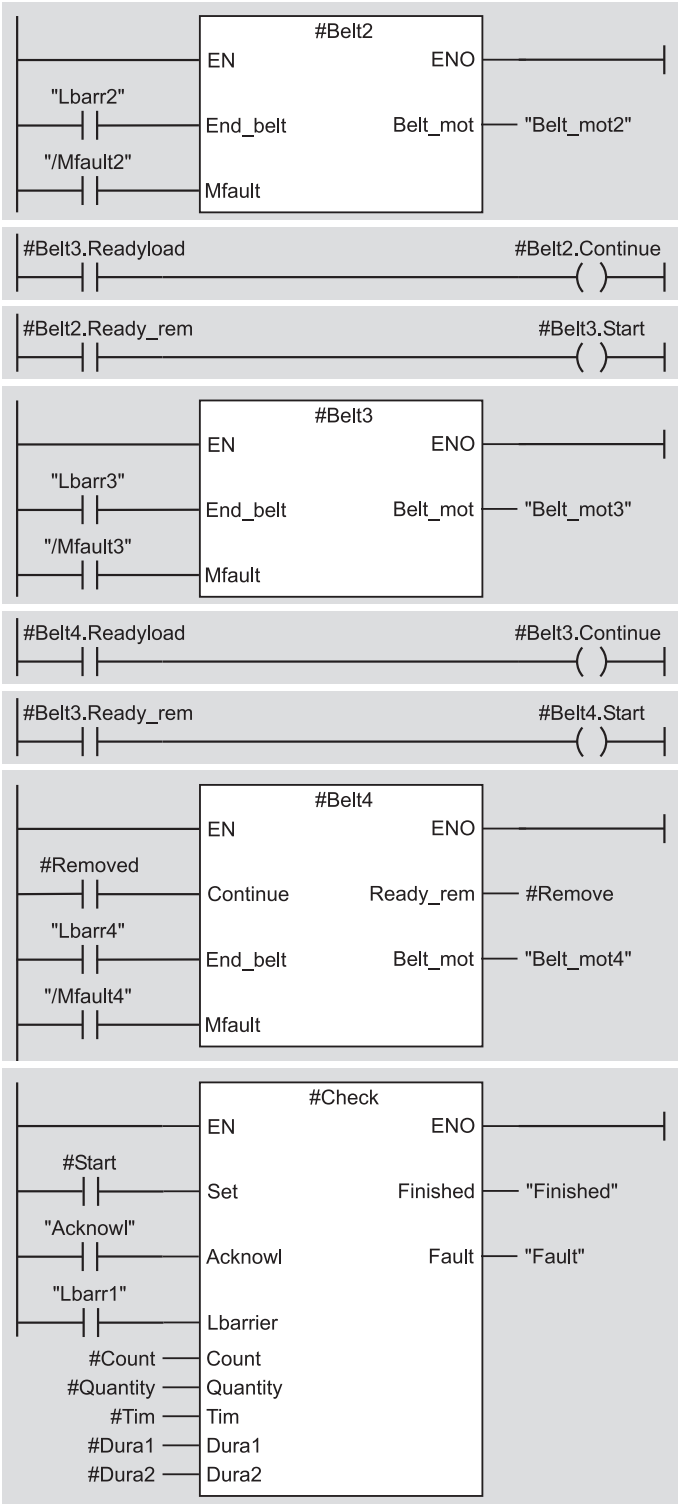


Figure 19.8 Program for Feed example (LAD)

Continued overleaf



Continued:

Figure19.8
Program for Feed Example
(LAD)

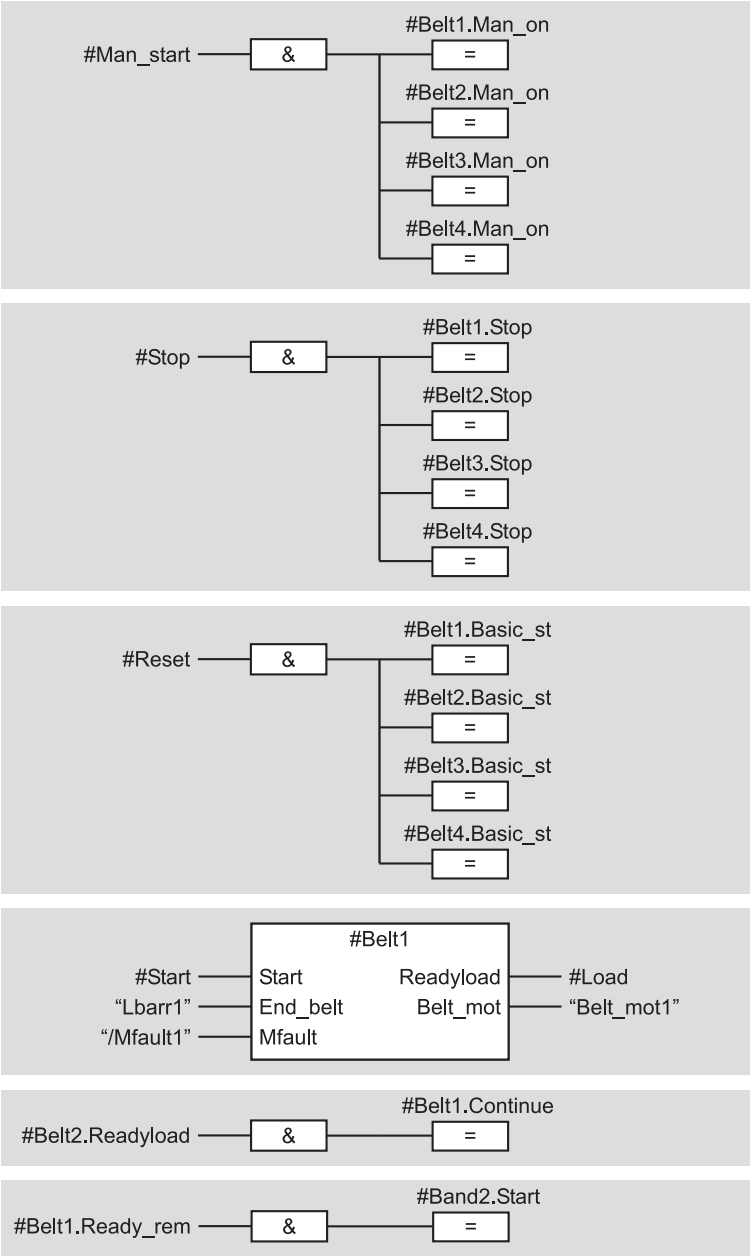
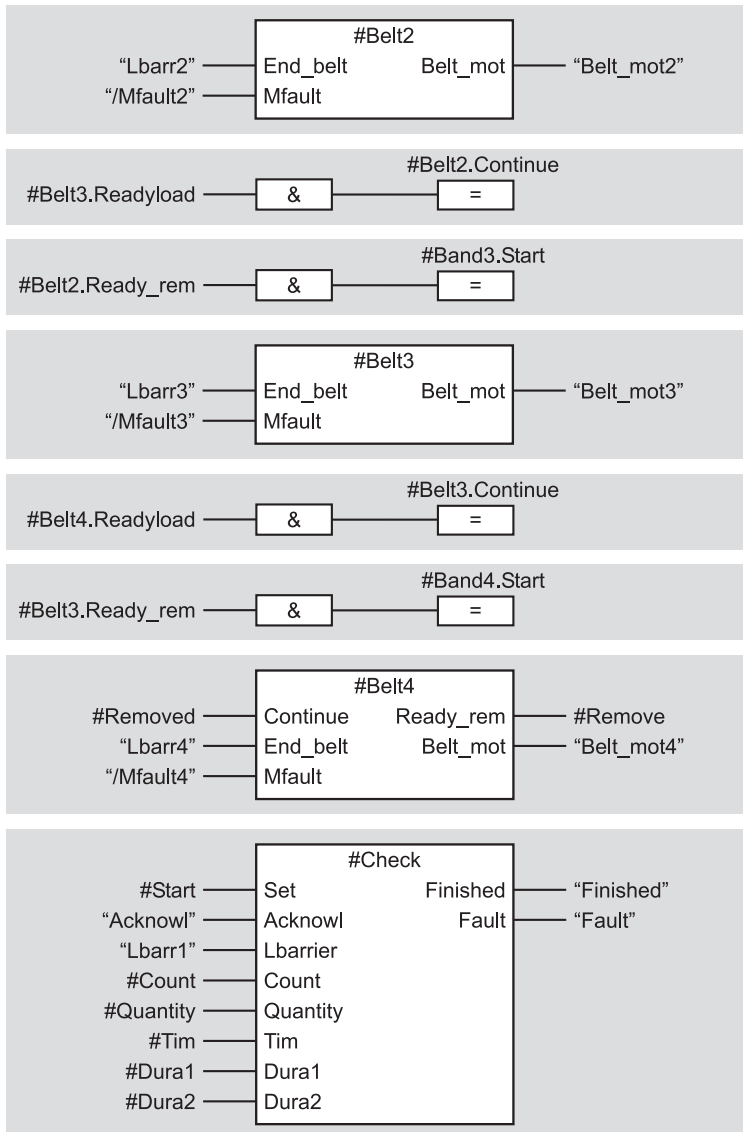


Figure 19.9 Program for Feed Example (FBD)

Continued overleaf



Continued

Figure 19.9 Program for Feed Example (FBD)

Table 19.6 Symbol Table for the Conveyor Belt, Parts Counter and Feed Examples

Symbol	Address	Data Type	Comment
Conveyor_belt	FB 21	FB 21	Conveyor belt control
Belt_data1	DB 21	FB 21	Data for conveyor belt 1
Belt_data2	DB 22	FB 21	Data for conveyor belt 2
Parts_counter	FB 22	FB 22	Counter control and monitor
CounterDat	DB 29	FB 22	Data for parts counter
Feed	FB 20	FB 20	Feed with several belts
FeedDat	DB 20	FB 20	Data for Feed
Cycle	OB 1	OB 1	Main program, cyclic execution
Basic_st	I 0.0	BOOL	Set controller to the basic state
Man_on	I 0.1	BOOL	Switch on conveyor belt motors
/Stop	I 0.2	BOOL	Stop conveyor belt motors (zero active)
Start	I 0.3	BOOL	Start conveyor belt
Continue	I 0.4	BOOL	Acknowledgment that parts have been removed
Acknowl	I 0.6	BOOL	Acknowledge fault
Set	I 0.7	BOOL	Set counter, activate monitor
Lbarr1	I 1.0	BOOL	(Light barrier) "End of belt" sensor signal for conveyor belt 1
Lbarr2	I 1.1	BOOL	(Light barrier) "End of belt" sensor signal for conveyor belt 2
Lbarr3	I 1.2	BOOL	(Light barrier) "End of belt" sensor signal for conveyor belt 3
Lbarr4	I 1.3	BOOL	(Light barrier) "End of belt" sensor signal for conveyor belt 4
/Mfault1	I 2.0	BOOL	Motor protection switch conveyor belt 1 (zero-active)
/Mfault2	I 2.1	BOOL	Motor protection switch conveyor belt 2 (zero-active)
/Mfault3	I 2.2	BOOL	Motor protection switch conveyor belt 3 (zero-active)
/Mfault4	I 2.3	BOOL	Motor protection switch conveyor belt 4 (zero-active)
Readyload	Q 4.0	BOOL	Load new parts onto belt
Ready_rem	Q 4.1	BOOL	Remove parts from belt
Finished	Q 4.2	BOOL	Number of parts reached
Fault	Q 4.3	BOOL	Monitor activated
Belt_mot1	Q 5.0	BOOL	Switch on belt motor for conveyor belt 1
Belt_mot2	Q 5.1	BOOL	Switch on belt motor for conveyor belt 2
Belt_mot3	Q 5.2	BOOL	Switch on belt motor for conveyor belt 3
Belt_mot4	Q 5.3	BOOL	Switch on belt motor for conveyor belt 4
Quantity	MW 4	WORD	Number of parts
Dura1	MW 6	S5TIME	Monitoring time for light barrier covered
Dura2	MW 8	S5TIME	Monitoring time for light barrier not covered
Count	C 1	COUNTER	Counter for parts
Monitor	T 1	TIMER	Timer for monitor

Program Processing

This section of the book discusses the various methods of program processing.

The **main program** executes cyclically. After each program pass, the CPU returns to the beginning of the program and executes it again. This is the “standard” method of processing PLC programs.

Numerous system functions support the utilization of system services, such as controlling the real-time clock or communication via bus systems. In contrast to the static settings made when parameterizing the CPU, system functions can be used dynamically at program run time.

The main program can be temporarily suspended to allow **interrupt servicing**. The various types of interrupts (time-of-day interrupts, time-delay interrupts, watchdog interrupts, hardware interrupts, DPV1 interrupts, synchronous cycle interrupts, multiprocessor interrupt) are divided into priority classes whose processing priority you may yourself, to a large degree, determine. Interrupt servicing allows you to react quickly to signals from the controlled process or implement periodic control procedures independently of the processing time of the main program.

Before starting the main program, the CPU initiates a **start-up program** in which you can make specifications regarding program processing, define default values for variables, or parameterize modules.

Error handling is also part of program processing. STEP 7 distinguishes between synchronous errors, which occur during processing of a statement, and asynchronous errors, which can be detected independently of program processing. In both cases you can adapt the error routine to suit your needs.

20 Main program

Program structure; scan cycle control; response time; program functions; multi-computing operation; data exchange with system functions; start information

21 Interrupt handling

Time-of-day interrupts; time-delay interrupts; watchdog interrupts; hardware interrupts; DPV1 interrupts; multiprocessor interrupt; synchronous cycle interrupts; handling interrupt events

22 Start-up characteristics

Power-up, memory reset, retentivity; cold restart; warm restart; ascertain module address; parameterize modules

23 Error handling

Synchronous errors (programming errors, access errors); handling synchronous error events; asynchronous errors; system diagnostics

20 Main Program

The main program is the cyclically scanned user program; cyclic scanning is the “normal” way in which programs execute in programmable logic controllers. The large majority of control systems use only this form of program execution. If event-driven program scanning is used, it is in most cases only in addition to the main program.

The main program is invoked in organization block OB 1. It executes at the lowest priority level, and can be interrupted by all other types of program processing. The user program is executed in RUN mode, which is set using the mode selector on the CPU’s front panel. With a toggle switch as mode selector, the position is RUN, with a key-operated switch, the positions are RUN and RUN-P. When in the RUN-P position, the CPU can be programmed via a programming device. In the RUN position, you can remove the key so that no one can change the operating mode without proper authorization; when the mode selector is at RUN, programs can only be read.

20.1 Program Organization

20.1.1 Program Structure

To analyze a complex automation task means to divide that task into smaller tasks or functions in accordance with the structure of the process to be controlled. You then define the individual tasks resulting from this dividing process by determining the functions and stipulating the interface signals to the process or to other tasks. This breakdown into individual tasks can be done in your program. In this way, the structure of your program corresponds to the division of the automation task.

A divided user program can be more easily configured, and can be programmed in sections (even by several people in the case of very large

user programs). And finally, but not lacking in importance, dividing the program simplifies both debugging and service and maintenance.

The structuring of the user program depends on its size and its function. A distinction is made between three different “methods”:

In a **linear program**, the entire main program is in organization block OB 1. Each current path is in a separate network. STEP 7 numbers the networks in sequence. When editing and debugging, you can reference every network directly by its number.

A **partitioned program** is basically a linear program which is divided into blocks. Reasons for dividing the program might be because it is too long for organization block OB 1 or because you want to make it more readable. The blocks are then called in sequence. You can also divide the program in another block the same way you would the program in organization block OB 1. This method allows you to call associated process-related functions for processing from within one and the same block. The advantage of this program structure is that, even though the program is linear, you can still debug and run it in sections (simply by omitting or adding block calls).

A **structured program** is used when the conceptual formulation is particularly extensive, when you want to reuse program functions, or when complex problems must be solved. Structuring means dividing the program into sections (blocks) which embody self-contained functions or serve a specific functional purpose and which exchange the fewest possible number of signals with other blocks. Assigning each program section a specific (process-related) function will produce easily readable blocks with simple interfaces to other blocks when programmed.

The LAD and FBD programming languages support structured programming through func-

tions with which you can create “blocks” (self-contained program sections). Chapter 3.2, “Blocks”, discusses the different kinds of blocks and their uses. You will find a detailed description of the functions for calling and ending blocks in Chapter 18, “Block Functions”. The blocks receive the signals and data to be processed via the call interface (the block parameters), and forward the results over this same interface. The options for passing parameters are described in detail in Chapter 19, “Block Parameters”.

20.1.2 Program Organization

Program organization determines whether and in what order the CPU will process the blocks which you have generated. To organize your program, you program block calls in the desired sequence in the supraordinate blocks. You should chose the order in which the blocks are called so that it mirrors the process-related or function-related division of the controlled plant.

Nesting depth

The maximum depth applies for a priority class (for the program in an organization block), and is CPU-dependent. On the CPU 314, for example, the nesting depth is eight, that is, beginning with one organization block (nesting depth 1), you can add seven more blocks in the “horizontal” direction (this is called “nesting”). If more blocks are called, the CPU goes to STOP with a “Block overflow” error. Do not forget to include system function block (SFB) calls and system function (SFC) calls when calculating the nesting depth.

A data block call, which is actually only the opening or selecting of a data area, has no effect on the nesting depth of blocks, nor is the nesting depth affected by calling several blocks in succession (linear block calls).

Practice-related program organization

In organization block OB 1, you should call the blocks in the main program in such a way as to roughly organize your program. A program can be organized on either a process-related or function-related basis.

The following points of discussion can give only a rough, very general view with the intention of giving the beginner some ideas on program structuring and on translating his control task into reality. Advanced programmers normally have sufficient experience to organize a program to suit the special control task at hand.

A **process-related program structure** closely follows the structure of the plant to be controlled. The individual program sections correspond to the individual parts of the plant or of the process to be controlled. Subordinate to this rough structure are the scanning of the limit switches and operator panels and the control of the actuators and display devices (in different parts of the plant). Bit memory or global data

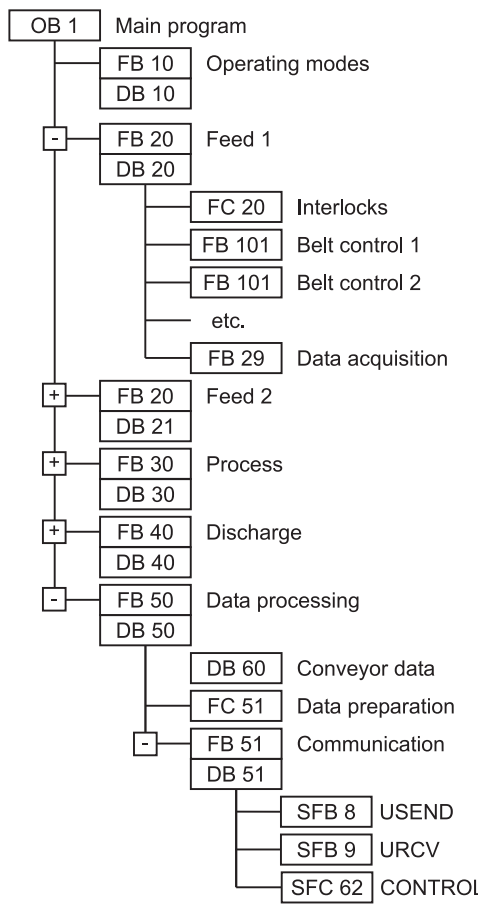


Figure 20.1 Example of Program Structuring

are used for signal interchange between different parts of the plant.

A **function-related program structure** is based on the control function to be executed. Initially, this method of program structuring does not take the controlled plant into account at all. The plant structure first becomes apparent in the subordinate blocks when the control function defined by the rough structure is divided further.

In practice, a hybrid of these two concepts is normally used. Figure 20.1 shows an example: A functional structure is mirrored in the operating mode program and in the data processing program which goes above and beyond the plant itself. Program sections Feeding Conveyor 1, Feeding Conveyor 2, Process and Discharging Conveyor are process-related.

The example also shows the use of different types of blocks. The main program is in OB 1; it is in this program that the blocks for the operating modes, the various pieces of plant equipment, and for data processing are called. These blocks are function blocks with an instance data block as data store. Feeding Conveyor 1 and Feeding Conveyor 2 are identically structured; FB 20, with DB 20 as instance data block for Feeding Conveyor 1 and with DB 21 as instance data block for Feeding Conveyor 2, is used for control.

In the conveyor control program, function FC 20 processes the interlocks; it scans inputs or memory bits and controls FB 20's local data. Function block FB 101 contains the control program for a conveyor belt, and is called once for each belt. The call is a local instance, so that its local data are in instance data block DB 20. The same applies for the data acquisition program in FB 29.

The data processing program in FB 50, which uses DB 50, processes the data acquired with FB 29 (and other blocks), which are located in global data block DB 60. Function FC 51 prepares these data for transfer. The transfer is controlled by FB 51 (with DB 51), in which system blocks SFB 8, SFB 9 and SFB 62 are called. Here, too, the SFBs save their instance data in "supraordinate" data block DB 51.

20.2 Scan Cycle Control

20.2.1 Process Image Updating

The process image is part of the CPU's internal system memory (Chapter 1.1.6, "CPU Memory Areas"). It begins at I/O address 0 and ends at an upper limit stipulated by the CPU. On appropriately equipped CPUs, you can define this limit yourself.

Normally, all digital modules lie in the process image address area, while all analog modules have addresses outside this area. If the CPU has free address allocation, you can use the configuration table to direct any module over the process image or address it outside the process image area.

The process image consists of the process-image input table (inputs I) and the process-image output table (outputs Q).

After CPU restart and prior to the first execution of OB 1, the operating system transfers the signal states of the process-image output table to the output modules and accepts the signal states of the input modules into the process-image input table. This is followed by execution of OB 1 where normally the inputs are combined with each other and the outputs are controlled. Following termination of OB 1, a new cycle begins with the updating of the process image (Figure 20.2).

If an error occurs during automatic updating of the process image, e.g. because a module is no longer accessible, organization block OB 85 "Program Execution Errors" is called. If OB 85 is not available, the CPU goes to STOP.

Subprocess images

With appropriately equipped CPUs, you can divide the process image into up to 30 subprocess images. You make this division during parameterization of the signal modules by defining the subprocess image via which the module is to be addressed when you assign addresses. You can separate the division according to process-image input table and process-image output table.

All modules that you do not assign to one of the subprocess images 1 to 30 are stored in subprocess image 0, which is also called the OB1 pro-

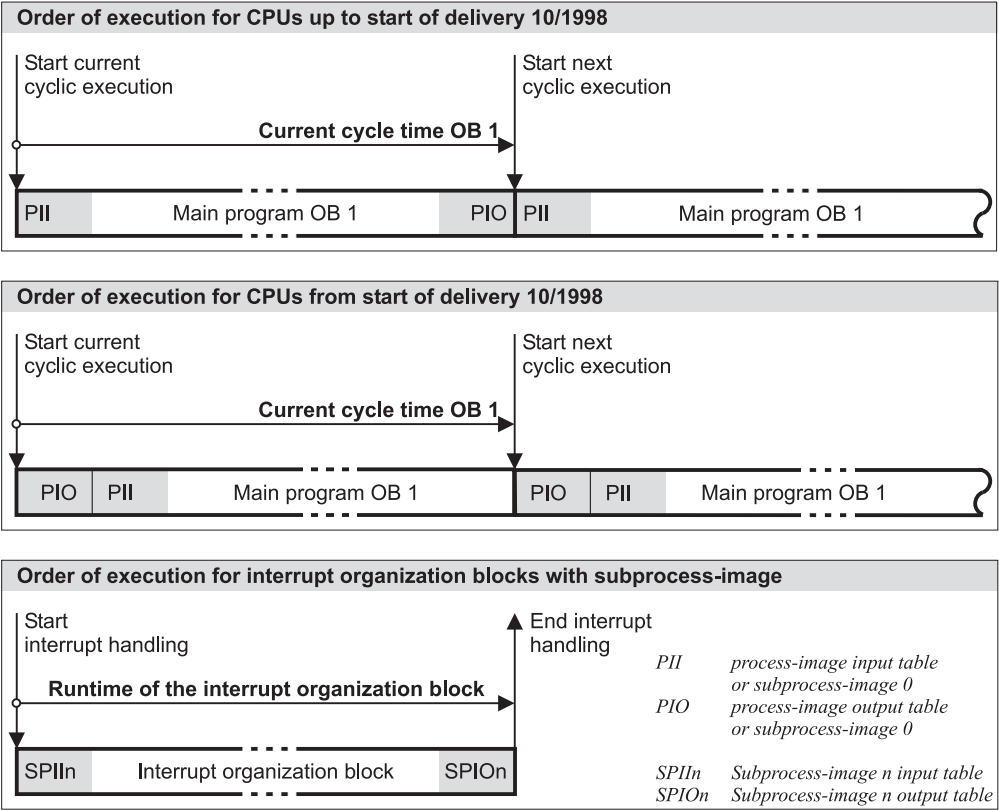


Figure 20.2 Updating the Process Image

cess image (OB1-PI). This subprocess image 0 is updated automatically by the operating system of the CPU as part of cyclic execution. You can also program the CPU properties for S7-400 such that the automatic updating is switched off.

With appropriately equipped CPUs, you can also assign the subprocess images to the interrupt organization blocks so that they are automatically updated when these OBs are called.

The system functions SFC 26 UPDAT_PI and SFC 27 UPDAT_PO are available for updating the subprocess images by the user program. In the organization blocks for synchronous cycle interrupt, you use the system functions SFC 126 SYNC_PI and SFC 127 SYNC_PO (Chapter 21.8.2, "Isochrone Updating Of Process Image").

SFC 26 UPDAT_PI SFC 27 UPDAT_PO

The system function SFC 26 UPDAT_PI updates a subprocess image of the inputs, the system function SFC 27 UPDAT_PO updates a subprocess image of the outputs. Table 20.1 shows the parameters of these SFCs. You can also update subprocess image 0 with these SFCs.

You can carry out updating of individual subprocess images by calling these SFCs at any time and at any location. For example, you can define a subprocess image for a priority class (a program execution level) and you can then cause this subprocess image to be updated at the start and at the end of the relevant organization block when this priority class is processed.

Updating of a process image can be interrupted by calling a higher priority class. If an error

occurs during updating of a process image, e.g. because a module can no longer be accessed, this error is reported via the function value of the SFC.

20.2.2 Scan Cycle Monitoring Time

Program scanning in organization block OB 1 is monitored by the so-called “scan cycle monitor” or “scan cycle watchdog”. The default value for the scan cycle monitoring time is 150 ms. You can change this value in the range from 1 ms to 6 s by parameterizing the CPU accordingly.

If the main program takes longer to scan than the specified scan cycle monitoring time, the CPU calls OB 80 (“Timing error”). If OB 80 has not been programmed, the CPU goes to STOP.

The scan cycle monitoring time includes the full scan time for OB 1. It also includes the scan times for higher priority classes which interrupt the main program (in the current cycle). Communication processes carried out by the operating system, such as GD communication or PG access to the CPU (block status!), also increase the runtime of the main program. The increase can be reduced in part by the way you parameterize the CPU (“Cyclic load from communication” on the “Cycle/Clock memory bits” tab).

Cycle statistics

If you have an online connection from a programming device to an operating CPU, select PLC → MODULE INFORMATION to call up a dialog box that contains several tabs. The “Cycle Time” tab shows the current cycle time as well as the shortest and longest cycle time.

The parameterized minimum cycle duration and the scan cycle monitoring time are also displayed.

The cycle time for the last cycle and the minimum and maximum cycle time since the PLC was last started up can also be read in the temporary local data in the start information of OB 1.

SFC 43 RE_TRIGR Restarting the scan cycle monitoring time

An SFC 43 RE_TRIGR system function call restarts the scan cycle monitoring time; the timer restarts with the new value set via CPU parameterization. SFC 43 has no parameters.

Operating system run times

The scan cycle time also includes the operating system run times. These are composed of the following:

- ▷ System control of cyclic scanning (“no-load cycle”), fixed value
- ▷ Updating of the process image; dependent on the number of bytes to be updated
- ▷ Updating of the timers; dependent on the number of timers to be updated
- ▷ Communications load

Communications functions for the CPU include the transfer of user program blocks or data exchange between CPU modules using system functions. The time the CPU is to use for these functions can be limited by parameterizing the CPU.

All values at operating system runtime are properties of the relevant CPU.

Table 20.1 Parameters for the SFCs for Process Image Updating

Parameter Name	SFC		Declaration	Data Type	Contents, Description
PART	26	27	INPUT	BYTE	Number of the subprocess image (0 to 15)
RET_VAL	26	27	RETURN	INT	Error information
FLADDR	26	27	OUTPUT	WORD	On an access error: the address of the first byte to cause the error

20.2.3 Minimum Scan Cycle Time, Background Scanning

With appropriately equipped CPUs, you may specify a minimum scan cycle time. If the main program (including interrupts) takes less time, the CPU waits until the specified minimum scan cycle time has elapsed before beginning the next cycle by recalling OB 1.

The default value for the minimum scan cycle time is 0 ms, that is to say, the function is disabled. You can set a minimum scan cycle time of from 1 ms to 6 s in “Cycle/Clock memory bits” tab when you parameterize the CPU.

Background scanning OB 90

In the interval between the actual end of the cycle and expiration of the minimum cycle time, the CPU executes organization block OB 90 “Background scanning” (Figure 20.3). OB 90 is executed “in slices”. When the operating system calls OB 1, execution of OB 90 is interrupted; it is then resumed at the point of interruption when OB 1 has terminated. OB 90 can be interrupted after each statement, any system

block called in OB 90, however, is first scanned in its entirety.

The length of a “slice” depends on the current scan cycle time of OB 1. The closer OB 1’s scan time is to the minimum scan cycle time, the less time remains for executing OB 90. The program scan time is not monitored in OB 90.

OB 90 is scanned only in RUN mode. It can be interrupted by interrupt and error events, just like OB 1. The start information in the temporary local data (Byte 1) also tells which events cause OB 90 to execute from the beginning:

- ▷ B#16#91
After a CPU restart;
- ▷ B#16#92
After a block processed in OB 90 was deleted or replaced;
- ▷ B#16#93
After (re)loading of OB 90 in RUN mode;
- ▷ B#16#95
After the program in OB 90 was scanned and a new background cycle begins.

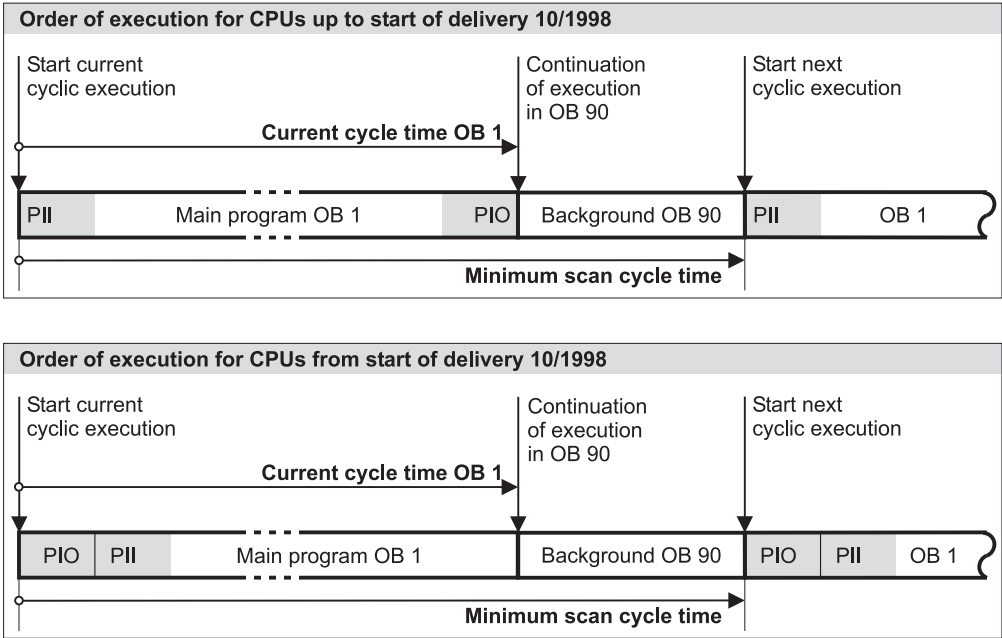


Figure 20.3 Minimum Cycle Duration and Background Scanning

20.2.4 Response Time

If the user program in OB 1 works with the signal states of the process images, this results in a response time which is dependent on the program execution time (scan cycle time). The response time lies between one and two scan cycles, as the following example explains.

When a limit switch is activated, for instance, it changes its signal state from “0” to “1”. The programmable controller detects this change during the subsequent updating of the process image, and sets the inputs allocated to the limit switch to “1”. The program evaluates this change by resetting an output, for example, in order to switch off the corresponding motor. The new signal state of the output that was reset is transferred at the end of the program scan; only then is the corresponding bit reset on the digital output module.

In a best-case situation, the process image is updated immediately following the change in the limit switch's signal (Figure 20.4). It would then take only one cycle for the relevant output to respond. In the worst-case situation, updating of the process image was just completed when the limit switch signal changed. It would then be necessary to wait approximately one cycle for the programmable controller to detect the signal change and set the input. After yet another cycle, the program can respond.

When so considered, the user program's execution time contains all procedures in one program cycle (including, for instance, the servic-

ing of interrupts, the functions carried out by the operating system, such as updating timers, controlling the MPI interface and updating the process images).

The response time to a change in an input signal can thus be between one and two cycles. Added to the response time are the delays for the input modules, the switching times of contactors, and so on.

In some instances, you can reduce the response times by addressing the I/Os directly or calling program sections on an event-driven basis (process interrupt).

You can achieve uniform response times or equal time intervals for the process control if a program component is always processed at the same time interval, e.g. a cyclic interrupt program. Program processing isochronous with the processing cycle of a PROFIBUS DP master system also produces calculable response times (Chapter 21.8, “Synchronous Cycle Interrupts”).

20.2.5 Start Information

The CPU's operating system forwards start information to organization block OB 1, as it does to every organization block, in the first 20 bytes of temporary local data. You can generate the declaration for the start information yourself or you can use information from *Standard Library* under *Organization Blocks*.

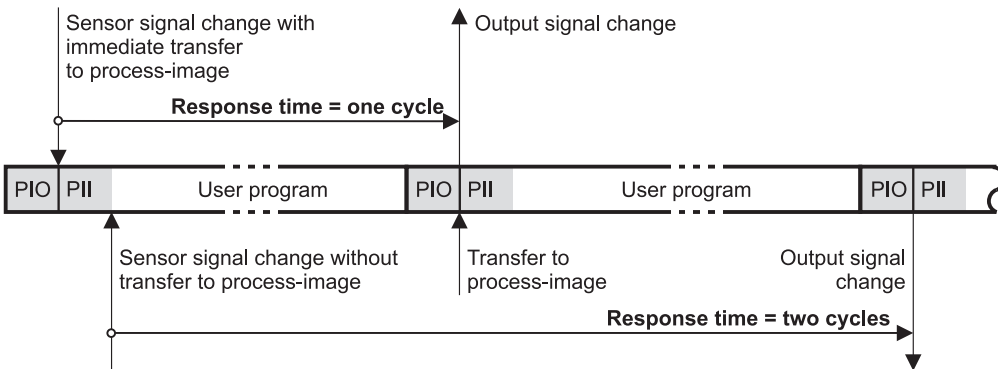


Figure 20.4 Response Times of Programmable Controllers

Table 20.2 Start Information for the OB 1

Byte	Variable Name	Data Type	Description	Contents
0	OB1_EV_CLASS	BYTE	Event class	B#16#11 = Call standard OB
1	OB1_SCAN_1	BYTE	Start information	B#16#01 = 1st cycle after complete restart B#16#02 = 1st cycle after warm restart B#16#03 = Every other cycle B#16#04 = 1st cycle after cold restart
2	OB1_PRIORITY	BYTE	Priority	B#16#01
3	OB1_OB_NUMBR	BYTE	OB Number	B#16#01
4	OB1_RESERVED_1	BYTE	Reserved	-
5	OB1_RESERVED_2	BYTE	Reserved	-
6..7	OB1_PREV_CYCLE	INT	Previous scan cycle time	in ms
8..9	OB1_MIN_CYCLE	INT	Minimum scan cycle time	in ms
10..11	OB1_MAX_CYCLE	INT	Maximum scan cycle time	in ms
12..19	OB1_DATE_TIME	DT	Event occurrence	Call time of the OB (cyclic)

Table 20.2 shows this start information for the OB1, the default symbolic designation, and the data types. You can change the designation at any time and choose names more acceptable to you. Even if you don't use the start information, you must reserve the first 20 bytes of temporary local data for this purpose (for instance in the form of a 20-byte array).

In SIMATIC S7, all event messages have a fixed structure which is specified by the event class. The start information for OB 1, for instance, reports event B#16#11 as a standard OB call. From the contents of the next byte you can tell whether the main program is in the first cycle after power-up and is therefore calling, for instance, initialization routines in the cyclic program.

The priority and OB number of the main program are fixed. With three INT values, the start information provides information on the cycle time of the last scan cycle and on the minimum and maximum cycle times since the last power-up. The last value, in DATE_AND_TIME format, indicates when the priority control program received the event for calling OB 1.

Note that direct reading of the start information for an organization block is possible only in that

organization block because that information consists of temporary local data. If you require the start information in blocks which lie on deeper levels, call system function SFC 6 RD_SINFO at the relevant location in the program.

SFC 6 RD_SINFO

Reading out start information

System function SFC 6 RD_SINFO makes the start information on the current organization block (that is, the OB at the top of the call tree) and on the start-up OB last executed available to you even at a deeper call level (Table 20.3).

Output parameter TOP_SI contains the first 12 bytes of start information on the current OB, output parameter START_UP_SI the first 12 bytes of start information on the last start-up OB executed. There is no time stamp in either case.

SFC 6 RD_SINFO can not only be called at any location in the main program but in every priority class, even in an error organization block or in the start-up routine. If the SFC is called in an interrupt organization block, for example, TOP_SI contains the start information of the interrupt OB. In the case of a call at restart, TOP_SI and START_UP_SI have the same contents.

Table 20.3 Parameters for SFC 6 RD_SINFO

SFC	Parameter Name	Declaration	Data Type	Contents, Description
6	RET_VAL	RETURN	INT	Error information
	TOP_SI	OUTPUT	STRUCT	Start information for the current OB (with the same structure as START_UP_SI)
	START_UP_SI	OUTPUT	STRUCT	Start information for the last OB started:
	.EV_CLASS		BYTE	Event ID and event class
	.EV_NUM		BYTE	Event number
	.PRIORITY		BYTE	Execution priority (number of the execution level)
	.NUM		BYTE	OB number
	.TYP2_3		BYTE	ID of supplementary information 2_3
	.TYP1		BYTE	ID of supplementary information 1
	.ZI1		WORD	Supplementary information 1
	.ZI2_3		DWORD	Supplementary information 2_3

20.3 Program Functions

In addition to parameterizing the CPU with the Hardware Configuration, you can also select a number of program functions dynamically at runtime via the integrated system functions.

20.3.1 Time of day

Each SIMATIC CPU has a clock which you can set and scan using STEP 7 or system functions. The time of day is represented in the user program in the format DATE_AND_TIME, consisting of the date, time and day of week.

Newer CPUs with firmware version 3 and higher also have the time status. You can then additionally set a difference from a time zone as well as summer/winter time identification.

The following system functions can be used to access the CPU clock:

- ▷ SFC 0 SET_CLK
Set date and time
- ▷ SFC 1 READ_CLK
Read date and time
- ▷ SFC 48 SNC_RTCB
Synchronize CPU clocks
- ▷ SFC 100 SET_CLKS
Set time of day, date and time status

You will find a list of system function parameters in Table 20.4.

Setting and reading the time of day

Calling the SFC 0 SET_CLK or the SFC 100 SET_CLKS using MODE = B#16#01 or B#16#03 set the clock to the value defined by the PDT parameter. SFC 0 SET_CLK sets the winter time for CPUs with summer/winter time identification. With the SFC 100 SET_CLKS, you can use the SUMMER parameter to define whether the time is the winter time (with “0”) or the summer time (with “1”).

The current time is read by SFC 1 READ_CLK and output in the CDT parameter. The time has the format DATE_AND_TIME when setting and reading, i.e. therefore contains the date and time.

Module time, local time

The time present on the CPU is the module time. This is decisive for all timed processes which control the CPU, such as run-time meters, starting of time-of-day interrupts or entering of time stamps in the diagnostics buffer and OB start information. You can set and read the module time using the system functions for the CPU clock.

Correspondingly designed CPUs additionally save a “time status”. This contains a correction value which results in the local time when added to the module time. The correction value is set in intervals of 30 minutes, and can also be negative (CORR parameter of SFC 100 SET_CLKS).

The local time can be used to display time zones.

Table 20.4 SFC Parameters for the CPU Clock

SFC	Parameter Name	Declaration	Data Type	Contents, Description
0	PDT	INPUT	DT	Date and time (new)
	RET_VAL	RETURN	INT	Error information
1	RET_VAL	RETURN	INT	Error information
	CDT	OUTPUT	DT	Date and time (current)
48	RET_VAL	RETURN	INT	Error information
100	MODE	INPUT	BYTE	Operating mode B#16#01: Only set time B#16#02: Only set time status B#16#03: Set time and time status
	PDT	INPUT	DT	Defined time
	CORR	INPUT	INT	Difference from basic time in 0.5-hour intervals from -24 to +26
	SUMMER	INPUT	BOOL	Summer/winter time identification ("1" = summer time)
	ANN_1	INPUT	BOOL	Announcement of time switchover: a "1" indicates that a switch is made from summer time to winter time or vice versa the next time that the hour changes
	RET_VAL	RETURN	INT	Error information

Time status

The time status is set when parameterizing the CPU with STEP 7 or with the SFC 100 SET_CLKS. The time of day and the time status can be read using the SFC 51 RDSYSST via the system status list (SSL_ID = W#16#0132 with INDEX W#16#0008). The *status* variable includes:

- ▷ The correction value (bits 2 to 6) in the 30-minute interval
- ▷ The sign of the correction value (bit 7)
- ▷ The summer/winter time identification (bit 14)
- ▷ The announcement hour (bit 15)

The *summer/winter time identification* shows whether the local time calculated from the module time and the correction value is the summer time (with "1") or the winter time (with "0").

If the *announcement hour* bit has the status "1", the switchover from summer time to winter time is carried out at the next change in hour.

Using the time status information, a local time can be generated from the module time in order to control timed processes in the user program.

Loadable standard blocks help you to handle the summer/winter time switching of the local time in the user program, in particular the starting of time-of-day interrupts depending on the local time (see further below under, "Loadable time-of-day blocks").

Time synchronization

In an automation network with several SIMATIC stations exchanging data with one another on subnets, it is possible to synchronize the clocks of all CPUs. You parameterize the clock of one CPU as the "Master clock", and set the interval at which the synchronization is to be carried out. You parameterize the blocks to be synchronized as "Slave clocks".

The synchronization can be carried out within an S7 station over the communications bus (backplane bus) or between stations over the MPI bus. This is carried out at the parameterized interval automatically when the master clock is set for the first time. If you set a master clock with the SFC 0 SET_CLK or SFC 100 SET_CLKS, all other clocks in the subnetwork are automatically synchronized to this value.

By calling the SFC 48 SNC_RTCB in the master clock, you can synchronize all slave blocks independent of the automatic interval.

If the master clock does not have a time-of-day status, the slave clocks are synchronized with the winter time. The correction factor is zero, the local time then corresponds to the module time.

If the master clock works with a time-of-day status, the complete time status is transmitted in addition to the time. Therefore the same local time (the same time zone) exists on all CPUs in the time network.

Set time using STEP 7

When setting the CPU parameters, you can set the synchronization mode (master, slave or none) and the synchronization interval in the attributes window on the “Diagnostics/clock” tab). The correction value set here is used to set the clock accuracy.

You can set the time and the time status using STEP 7 if the programming device is connected online to a CPU. Select PLC → DIAGNOSTICS/CUSTOMIZE → SET TIME OF DAY. In the enhanced dialog, you can set the local time as a difference from the module time, and define the summer/winter time. The time status is shown in the “Status” box.

Loadable time-of-day blocks

The program *Miscellaneous Blocks* in the *Standard Library* contains loadable blocks for handling summer/winter time switching and the local time in the user program.

- ▷ FC 60 LOC_TIME
Determine local time
- ▷ FC 61 BT_LT
Convert module time into local time
- ▷ FC 62 LT_BT
Convert local time into module time
- ▷ FC 63 S_LTINT
Set time-of-day interrupt to local time
- ▷ FB 60 SET_SW
Switch over summer/winter time

- ▷ FB 61 SET_SW_S
Switch over summer/winter time with time status
- ▷ UDT 60 WS_RULES
Rules for switching over summer/winter time (e.g. time for switching over)

20.3.2 Read System Clock

A CPU's system clock starts running on power-up. The system clock keeps running as long as the CPU is executing the restart routine or is in RUN mode. When the CPU goes to STOP or HOLD, the current system time is “frozen”.

If you initiate a hot restart on an S7-400 CPU, the system clock starts running again using the saved value as its starting time. Cold restart or warm restart reset the system time.

The system time has data format TIME, whereby it can assume only positive values:

TIME#0ms to
TIME#24d20h31m23s647ms.

In the event of an overflow, the clock starts again at 0. Newer CPUs update the system clock every millisecond, older S7-300-CPU's every 10 milliseconds.

SFC 64 TIME_TCK Read system time

You can read the current system time with system function SFC 64 TIME_TCK. The RET_VAL parameter contains the system time in the TIME data format.

You can use the system clock, for example, to read out the current CPU runtime or, by computing the difference, to calculate the time between two SFC 64 calls. The difference between two values in TIME format is computed using DINT subtraction.

20.3.3 Run-Time Meter

A run-time meter in a CPU counts the hours. You can use the run-time meter for such tasks as determining the CPU runtime or ascertaining the runtime of devices connected to that CPU.

The value on the run-time meter is also retained following a cold restart, failure of the backup voltage, and following a memory reset.

Table 20.5 Parameters of the SFCs for the Run-Time Meter

SFC	Parameter	Declaration	Data Type	Contents, Description
2	NR	INPUT	BYTE	Number of the run-time meter (B#16#00 to B#16#07)
	PV	INPUT	INT	New value for the run-time meter
	RET_VAL	RETURN	INT	Error information
3	NR	INPUT	BYTE	Number of the run-time meter (B#16#00 to B#16#07)
	S	INPUT	BOOL	Start (with “1”) or stop (with “0”) run-time meter
	RET_VAL	RETURN	INT	Error information
4	NR	INPUT	BYTE	Number of the run-time meter (B#16#00 to B#16#07)
	RET_VAL	RETURN	INT	Error information
	CQ	OUTPUT	BOOL	Run-time meter running (“1”) or stopped (“0”)
	CV	OUTPUT	INT	Current value of the run-time meter
101	NR	INPUT	BYTE	Number of the run-time meter (B#16#00 to B#16#0F)
	MODE	INPUT	BYTE	Order code (see text)
	PV	INPUT	DINT	New value for the run-time meter
	RET_VAL	RETURN	INT	Error information
	CQ	OUTPUT	BOOL	Run-time meter running (“1”) or stopped (“0”)
	CV	OUTPUT	DINT	Current value of the run-time meter

The range of values and the number of run-time meters per CPU depend on the CPU. The range is 16 bits (2^{15} – 1 hours) or 32 bits (2^{31} – 1 hours). When the CPU is at STOP or HOLD, the run-time meter also stops running; when the CPU is restarted, the run-time meter must be restarted.

When a run-time meter reaches the maximum time duration, it stops and reports an overflow. A run-time meter can be set to a new value or reset to zero only via an SFC call.

The following system functions are available to control a run-time meter:

- ▷ SFC 2 SET_RTM
Set 16-bit run-time meter
- ▷ SFC 3 CTRL_RTM
Start or stop 16-bit run-time meter
- ▷ SFC 4 READ_RTM
Read 16-bit run-time meter
- ▷ SFC 101 RTM
Adjust 32-bit run-time meter

Table 20.5 shows the parameter for these system functions.

The NR parameter stands for the number of the run-time meter, and has the data type BYTE. It

can be initialized using a constant or a variable (as can all input parameters of elementary data type). The PV parameter (data type INT) is used to set the run-time meter to an initial value. SFC 3's-S-parameter starts (with signal state “1”) or stops (with signal state “0”) the selected run-time meter. CQ indicates whether the run-time meter was running (signal state “1”) or stopped (signal state “0”) when scanned. The CV parameter records the hours in INT format.

By assigning the MODE parameter of SFC 101, you can control a 32-bit run-time meter as follows:

- B#16#00 Read current meter value
- B#16#01 Start at last meter value
- B#16#02 Stop meter
- B#16#04 Set to value specified in PV
- B#16#05 Set and start at value specified in PV
- B#16#06 Set and stop at value specified in PV

You can also use the SFCs for a 16-bit run-time meter to control a 32-bit run-time meter. The

latter then responds as with a 16-bit range of values.

20.3.4 Compressing CPU Memory

Multiple deletion and reloading of blocks, which often occur during online block modification, can result in gaps in the CPU's work memory and in the RAM load memory which decrease the amount of usable space in memory. When you call the "Compress" function, you start a CPU program which fills these gaps by pushing the blocks together. You can initiate the "Compress" function via a programming device connected to the CPU or by calling system function **SFC 25 COMPRESS**. The parameters for SFC 25 are listed in Table 20.6.

The compression procedure is distributed over several program cycles. The SFC returns BUSY = "1" to indicate that it is still in progress, and DONE = "1" to indicate that it has completed the compression operation. The SFC cannot compress when an externally initiated compression is in progress, when the "Delete Block" function is active, or when PG functions are accessing the block to be shifted (for instance the Block Status function).

Note that blocks of a particular CPU-specific maximum length cannot be compressed, so that gaps would still remain in CPU memory. Only the Compress function initiated via the PG while the CPU is at STOP closes all gaps.

20.3.5 Waiting and Stopping

The system function **SFC 47 WAIT** halts the program scan for a specified period of time.

SFC 47 WAIT has input parameter WT of data type INT in which you can specify the waiting time in microseconds (μ s).

The maximum waiting time is 32767 μ s; the minimum waiting time corresponds to the execution time of the system function, which is CPU-specific.

SFC 47 can be interrupted by higher-priority events. On an S7-300, this increases the waiting time by the scan time of the higher-priority interrupt routine.

The system function **SFC 46 STP** terminates the program scan, and the CPU goes to STOP. SFC 46 STP has no parameters.

20.3.6 Multicomputing

The S7-400 enables multiprocessing. As many as four appropriately designed CPUs can be operated in one universal rack (UR) on the same I/O bus and C bus.

An S7-400 station is automatically in multicomputing if you arrange more than one CPU in the central rack in the Hardware Configuration. The slots are arbitrary; the CPUs are distinguished by a number assigned automatically in ascending order when the CPUs are plugged in. You can also assign this number yourself on the "Multicomputing" tab.

The configuration data for all the CPUs must be loaded into the PLC, even when you make changes to only one CPU.

After assigning parameters to the CPUs, you must assign each module in the station to a CPU. This is done by parameterizing the module in the "Addresses" tab under "CPU Allocation" (Figure 20.5).

At the same time that you assign the module's address area, you also assign the module's interrupts to this CPU. With VIEW → FILTER → CPU No. x-MODULES, you can emphasize the modules assigned to a CPU in the configuration tables.

Table 20.6 Parameters for SFC 25 COMPRESS

SFC	Parameter	Declaration	Data Type	Contents, Description
25	RET_VAL	RETURN	INT	Error information
	BUSY	OUTPUT	BOOL	Compression still in progress (with "1")
	DONE	OUTPUT	BOOL	Compression completed (with "1")

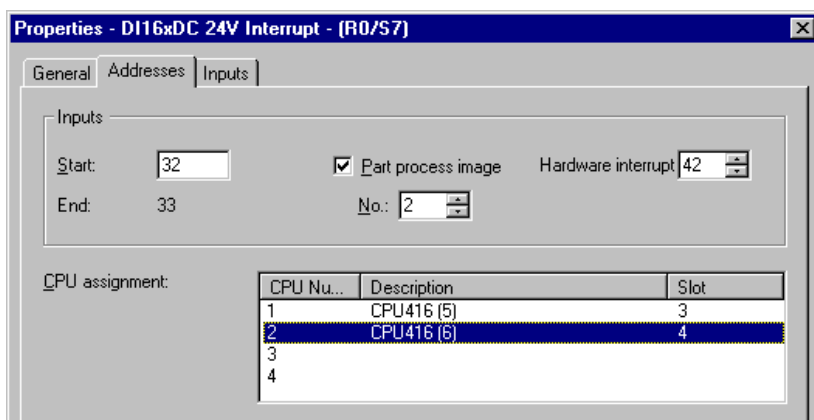


Figure 20.5 Module Assignments in Multicomputing

The CPUs in a multiprocessing network all have the same operating mode. This means

- ▷ They must all be parameterized with the same restart mode;
- ▷ They all go to RUN simultaneously;
- ▷ They all go to HOLD when you debug in single-step mode in one of the CPUs;
- ▷ They all go to STOP as soon as one of the CPUs goes to STOP.

When one rack in the station fails, organization block OB 86 is called in each CPU.

The user programs in these CPUs execute independently of one another; they are not synchronized.

An SFC 35 MP_ALM call starts organization block OB 60 “Multiprocessor interrupt” in all CPUs simultaneously (see Chapter 21.7, “Multiprocessor Interrupt”).

20.3.7 Determining the OB Program Runtime

The system function **SFC 78 OB_RT** determines the runtime of individual organization blocks over different time periods. This enables you to determine the time load (utilization) of the user program.

The operating system of a CPU designed for this purpose logs the runtimes of the individual organization blocks and makes them available for reading via the SFC 78 OB_RT. The accuracy

of the time acquisition is CPU-dependent and the times are specified in microseconds. If there is no value pending for a requested time, -1 (DW#16#FFFF FFFF) is returned.

Principle of time measurement

In the operating system of the CPU, a timer runs with a relative time in microseconds from 0 to $2^{31}-1$. At the transition from STOP to RUN, the timer is started, runs to the upper limit, and then starts again from zero.

The OB start event, the beginning and end of OB execution, and the interruptions caused by higher-priority OBs are captured in the operating system. The data of the last completed OB execution that was current at the time of calling the SFC 78 is stored.

SFC call outside the OB to be measured

When applying the SFC, a distinction is made between a call in the program of the requested OB and one outside the requested OB. Example: The SFC 78 is called in the OB 1 and is assigned a value of 30 in the parameter OB_NR. The last captured times for OB 30 are then read. Specification of the synchronous error OB with the numbers 121 and 122 is not permissible because these belong to the priority class of the error-causing OBs and thus to their program.

Figure 20.6 shows some examples of calling SFC 78 outside the OB to be measured. The ini-

Table 20.7 Parameters of the SFC 78

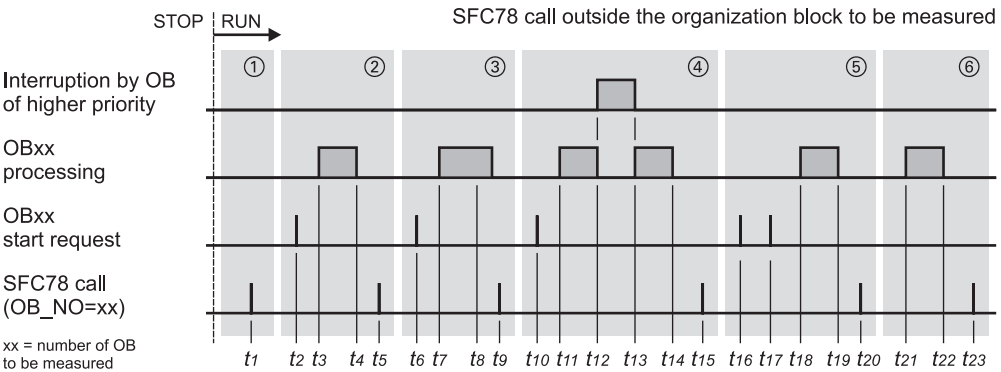
SFC	Parameter	Declaration	Data Type	Contents, Description
78	OB_NR	INPUT	INT	Number of the OB whose timers are to be scanned
	RET_VAL	RETURN	INT	Error information or number of the OB when OB_NR=0
	PRIO	OUTPUT	INT	Priority class of the scanned OB
	LAST_RT	OUTPUT	DINT	Runtime of the last completed execution
	LAST_ET	OUTPUT	DINT	Period between OB request and end of processing for the last completed execution
	CUR_T	OUTPUT	DINT	Period of OB request (relative time)
	CUR_RT	OUTPUT	DINT	Previous runtime of the OB execution
	CUR_ET	OUTPUT	DINT	Period between OB request and scanning by SFC 78 in the requested OB
	NEXT_ET	OUTPUT	DINT	Period since the next OB request and scanning by the SFC 78

tial values after a STOP-RUN transition are –1 (example ①).

LAST_RT indicates the runtime in microseconds of the last completed OB execution (examples ② to ⑥). The “net” runtimes are output. Interrupt times caused by OBs with higher

priority classes are not included in LAST_RT (④).

LAST_ET indicates the time period in microseconds between the start request and the end of processing for the last completed execution of



Values in the SFC78 parameters

Name	①	②	③	④	⑤	⑥
LAST_RT	–1	$t4 - t3$	$t8 - t7$	$(t14 - t13) + (t12 - t11)$	$t19 - t18$	$t22 - t21$
LAST_ET	–1	$t4 - t2$	$t8 - t6$	$t14 - t10$	$t19 - t16$	$t22 - t17$
CUR_T	–1	0	0	0	0	0
CUR_RT	–1	0	0	0	0	0
CUR_ET	–1	0	0	0	0	0
NEXT_ET	–1	–1	–1	–1	–1	–1

Figure 20.6 Calling SFC 78 outside the organization block to be measured

the OB to be measured (examples ② to ⑥). Interrupt times caused by higher priority classes are included in LAST_ET (④).

CUR_T indicates the relative time in microseconds (status of the counter in the operating system) of the start request of the OB. After initialization CUR_T (①) contains -1. On completion of OB execution, CUR_T is set to zero. Since SFC 78 is called outside the OB in these examples it consequently outputs zero at this parameter.

CUR_RT indicates the effective execution time of the OB until calling of SFC 78 in microseconds. After initialization CUR_RT (①) contains -1. After completion of OB execution, the value in CUR_RT is transferred to LAST_RT and CUR_RT is set to zero. Since the SFC call takes place outside the OB in these examples, the value is always zero.

CUR_ET indicates the time period from the OB start request to calling of SFC 78 in microseconds. After completion of OB execution, the value in CUR_ET is transferred to LAST_ET and CUR_ET is set to zero. Since the SFC call

takes place outside the OB in these examples, the value is always zero.

NEXT_RT indicates the time from the next OB start request to calling of the SFC in microseconds if further, unprocessed start requests are pending. In the case of the currently supplied CPUs, NEXT_RT is not calculated and the value is always -1.

SFC call in the program of the OB to be measured

SFC 78 can also be called in the program of the OB to be measured. The parameter OB_NR is then initialized with zero and the parameter RET_VAL returns the current OB number – in the case of error-free execution. The times for the OB in which SFC 78 is called are read. When calling in one of the synchronous error OBs with the numbers 121 and 122, this is the data of the OB causing the error.

Figure 20.7 shows some examples of calling SFC 78 in the program of the OB to be measured. This can be directly in the OB program

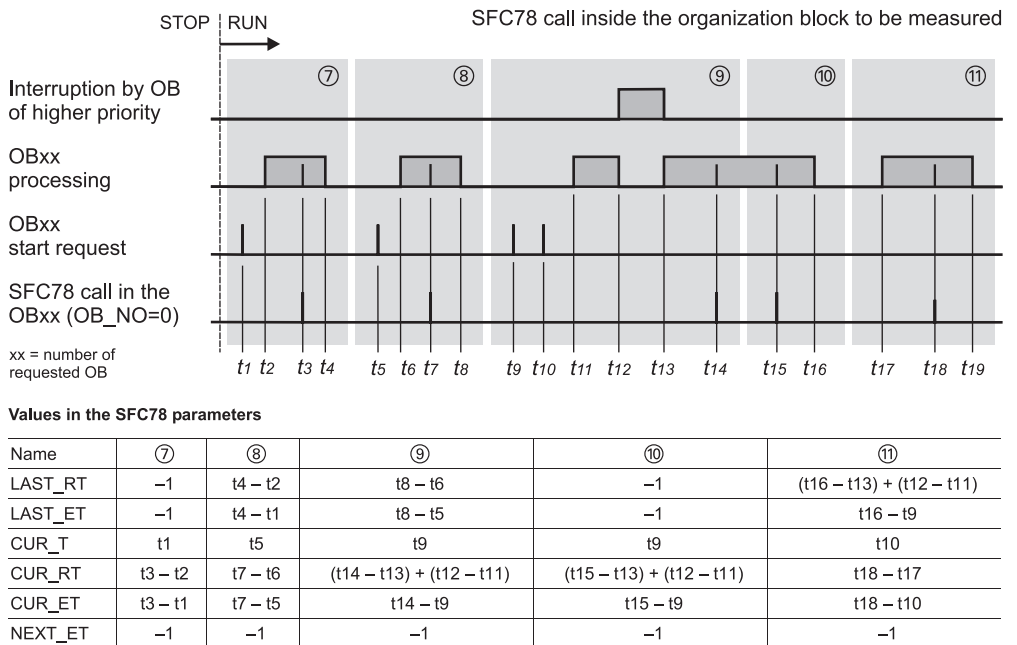


Figure 20.7 Calling SFC 78 within the organization block to be measured

Table 20.8 Parameters of SFC 109 PROTECT

SFC	Parameter	Declaration	Data type	Contents, Description
109	MODE	INPUT	WORD	Job ID: W#16#0000 Protection level 1 W#16#0001 Protection level 2
	RET_VAL	RETURN	INT	Error information

or in one of the blocks called there. The initial values after a STOP-RUN transition are –1.

LAST_RT indicates the runtime in microseconds of the last completed OB execution (examples ⑧, ⑨ and ⑪). If the SFC is called again in the OB to be measured, –1 is output (example ⑩). The “net” runtimes are output; interrupt times caused by higher priority classes are not included in LAST_RT (example ⑨).

LAST_ET indicates the time period in microseconds between the request and the end of processing for the last completed execution of the OB to be measured (examples ⑧ and ⑨). This also applies for the first call of SFC 78 in the OB to be measured (⑨). If the SFC is called again in the OB to be measured, –1 is output (example ⑩). LAST_ET also contains the interrupt times caused by higher priority classes (⑨).

CUR_T indicates the relative time in microseconds (status of the counter in the operating system) of the start request of the OBs, when – as in the following examples – the SFC 78 is called within the OB. On completion of OB execution, CUR_T is set to zero.

CUR_RT indicates the effective execution time of the OB in microseconds until calling of SFC 78. After completion of OB execution, the value in CUR_RT is transferred to LAST_RT and CUR_RT is set to zero. Interrupt times caused by higher priority classes are not included in CUR_RT (⑨ and ⑩).

CUR_ET indicates the time period from the OB start request to calling of SFC 78 in microseconds. After completion of OB execution, the value in CUR_ET is transferred to LAST_ET and CUR_ET is set to zero. CUR_ET also contains the runtimes of the higher-priority OBs that interrupt the OB currently to be measured.

NEXT_RT indicates the time from the next OB start request to calling of the SFC in microsec-

onds if further, unprocessed start requests are pending. In the case of the currently supplied CPUs, NEXT_RT is not determined and the value is always –1.

20.3.8 Changing program protection

The user program in a CPU can be protected against access in three protection levels (see Chapter 2.6.2, “Protecting the User Program”). Program-driven toggling between protection levels 1 and 2 is possible with the system function **SFC 109 PROTECT**. You can find the parameters of this system function in Table 20.8.

Calling SFC 109 PROTECT is only effective if you have set Protection level 1 with the hardware configuration. It remains ineffective if Protection level 2 or 3 is set, or if a password has been entered in Protection level 1 in conjunction with the option “Can be revoked by password”.

The protection level set with SFC 109 PROTECT remains unchanged if:

- ▷ the CPU goes to STOP due to a (program) error, an SFC 46 STP call, or operator intervention,
- ▷ the CPU is battery-backed and the mains supply is restored, or
- ▷ a restart is carried out (S7-400).

In all other cases, Protection level 1 is set in the case of an operating mode transition. Even if you switch the mode selector to STOP, Protection level 1 is (re)set.

You can determine the current protection level online in the SIMATIC Manager with selected CPU and PLC → DIAGNOSTIC/SETTING → OPERATING MODE. In the program, you can scan the protection level with the SFC 51 RDSYSST via the system status list W#16#0232 with the Index W#16#0004.

In the case of a CPU with key lock switch as mode selector, you can remove the key in the RUN position and permit read access only by the programming device during operation. In the case of a toggle switch as mode selector, this is not possible. In this case, you can call SFC 109 PROTECT in the restart organization blocks to activate Protection level 2, that is, write protection (read access only) when switching on the CPU.

With SFC 109 PROTECT, you can change the protection level during operation without operating the mode selector. Thus, for example, with SFC 109 PROTECT you can (re)set the protection level to 1 with `MODE = W#16#0000` depending on the signal state of a binary variable, in order, for example, to reload program sections. Then you re-activate write protection with `MODE = W#16#0001`.

20.4 Communication via Distributed I/O

Distributed I/O is understood to be modules connected over PROFIBUS DP or PROFINET IO.

With PROFIBUS DP, the DP master communicates with the DP slaves assigned to it over the PROFIBUS subnetwork. With PROFINET IO, it is the IO controller which exchanges data with the IO devices assigned to it over the Industrial Ethernet subnetwork.

The data transmission is carried out “automatically”, you do not need to be involved. You configure and address the distributed I/O using the Hardware Configuration tool in a manner similar to with the central modules. From the user program, you can address the inputs and outputs in the stations of the distributed I/O just like the inputs and outputs of the central modules.

20.4.1 Addressing PROFIBUS DP

The distributed modules (stations, DP slaves) with PROFIBUS DP are assigned to a DP master in a manner similar to how the central modules are assigned to a CPU and controlled by it. The DP master with all “its” DP slaves is

referred to as the DP master system. Several DP master systems can be present in an S7 station.

Newer DP masters can handle two operating modes: DPV1 and S7-compatible. “S7-compatible” corresponds to the previous mode. With this, you can operate all DP standard slaves according to EN 50170, and additionally the “DP S7 slaves” from Siemens which could already send interrupts to the DP master. In DPV1 mode, you can additionally use DP slaves which exhibit the new properties according to IEC 61131, such as increased diagnostics and parameterization capabilities resulting from the acyclic transmission of data records or the use of new types of interrupt. New system functions for transmission of data records, as well as new interrupt organization blocks, exist for these new “DPV1 slaves”.

Like the central modules, the DP slaves occupy addresses in the I/O area of the CPU (“logical address area”). The DP master is as it were “transparent” to the addresses of the DP slaves; the CPU “sees” the addresses of the DP slaves, meaning that these must not overlap with those of the central modules, not even with those of DP slaves in other DP master systems assigned to the CPU.

Every DP slave has three addresses in addition to the node address: a geographical address, a module starting address and at least one diagnostics address (Figure 20.8).

Node address

Every node on the PROFIBUS subnetwork has a unique address, the node address (station number) in that subnetwork that distinguishes it from the other nodes on the subnetwork. The station (the DP master or a DP slave) is accessed on PROFIBUS with this node address.

Please note that there must be a gap of at least 1 between the addresses of the active bus nodes (e.g. in the case of DP master and nodes in cross traffic). STEP 7 takes this into account when assigning node addresses automatically.

Geographical address

The geographical address identifies a module slot. With central modules, the geographical

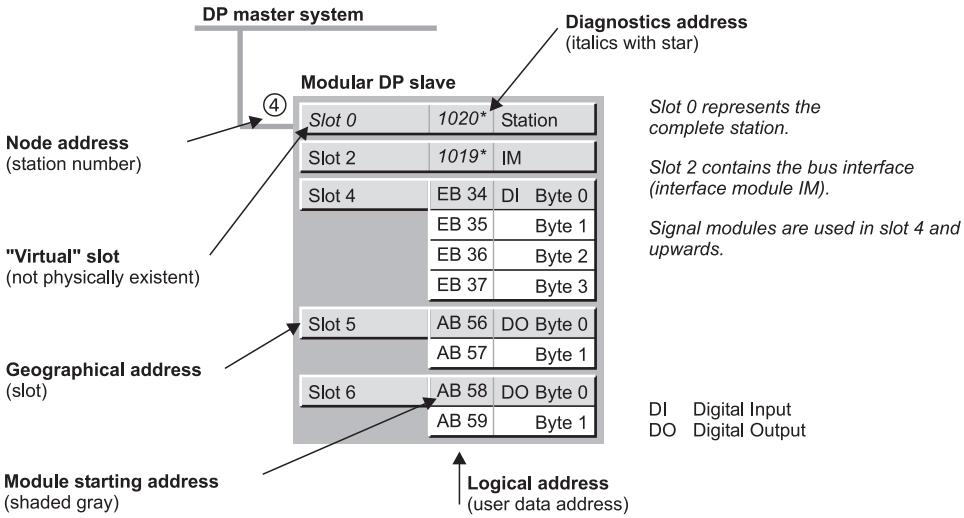


Figure 20.8 Addresses in a DP Master System

address contains the number of the rack and that of the slot. With PROFIBUS DP, the geographical address contains the number of the DP master system, the station number and the slot number.

Slot numbering of a DP slave depends on its type. If it is integrated using a GSD file, the entries in the GSD file determine the slot at which the I/O modules start. In the case of DP standard slaves, the slots for I/O modules begin at 1. Slot numbering of an S7 slave is based on the slots of an S7-300 station. Slots 1 (power supply) and 3 (expansion interface module) remain empty. Slot 2 (CPU) corresponds to the interface module (header module) of the modular DP slaves. The signal modules (SM) are positioned starting at slot 4. In addition, there is the “virtual” slot 0 (not physically present); this represents the complete station.

It is similar with intelligent DP slaves. In this case, the transfer memory is the interface to the DP master. Configuration of the transfer memory – which you carry out with the Hardware Configuration tool – generates areas which correspond to modules or slots. These slots do not really exist, and one therefore speaks of “virtual” slots.

Virtual slot 0 represents the DP station, virtual slot 2 the bus interface, in this case the slave

CPU as the “header module” of the DP slave. From virtual slot 4 onwards, the user data areas are present in the transfer memory; they correspond to the signal modules. The virtual slots in the transfer memory are “seen” by both the master and slave CPUs.

The definition of virtual slots makes it possible to directly assign diagnostics and interrupt events of the interface module or the station (see further below “Diagnostics address”). The system functions SFC 5 GADR_LGC and SFC 49 LGC_GADR are available for conversion from the geographical address to the logical addresses and vice versa, and the system functions SFC 70 GEO_LOG and SFC 71 LOG_GEO are also suitable for PROFINET IO.

Note that DP slaves which are incorporated into the hardware configuration through a GSD file according to EN 50170 of version 3 or later (DPV1) can save the user data starting at slot 1.

Logical address, module starting address

You use the logical address to access the user data of a station. Each byte of the user data is unambiguously identified by a logical address. The logical address corresponds to the absolute address; a symbol (name) can be assigned to it so that it is easier to read (symbolic addressing).

The smallest logical address of a module or station is the module starting address (see also Chapter 1.4, “Module Addresses”).

Diagnostics address

Modules and stations which can deliver diagnostics data and do not have a user data address themselves are identified by the diagnostics address. The diagnostics address occupies one byte of I/O input in the logical address volume. In the default setting, STEP 7 assigns the diagnostics address starting with the highest address in the I/O area of the CPU. You can change the diagnostics address. The address overview in the Hardware Configuration tool identifies the diagnostics address by a star.

Signal modules or user data areas in the transfer memory of intelligent DP slaves have logical addresses, also for scanning diagnostics data. The complete station delivers its diagnostics data via one diagnostics address which is assigned to virtual slot 0. With modular and intelligent DP slaves, the bus interface can deliver its diagnostics data via the diagnostics address of slot 2.

Figure 20.9 shows an example of the diagnostics addresses in a DP master system. A compact DP slave possesses one diagnostics address for the complete station, a modular DP slave possesses one diagnostics address for the station and one for the interface module. With intelligent DP slaves, there is also a diagnostics address for the DP interface.

The diagnostics addresses are assigned in descending order starting with the highest I/O address. For example, the DP interface of the CPU 317-2PN/DP is assigned the diagnostic address 8191, the PN interface and the two ports the addresses 8190 to 8188 (not shown in Figure 20.9), and the virtual slots 0 and 2 in the transfer memory the addresses 8187 and 8186. It is similar with the master CPU: the diagnostics addresses start at 16383 for the DP interface and continue with 16382 for the MPI/DP interface (not shown), 16381 and 16380 for the virtual slots 0 and 2 of the first intelligent DP slave, 16379 and 16378 for the second DP slave etc. The diagnostics addresses for the DP slaves are assigned by the Hardware Configuration

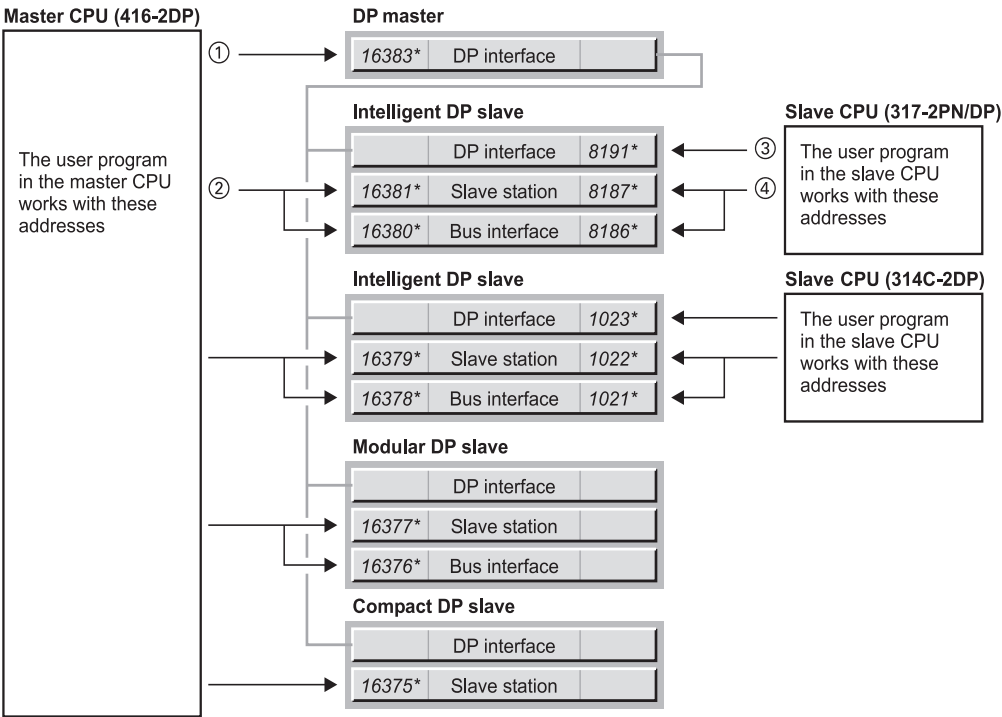
tool in the sequence of coupling to the DP master system.

In the user program, the diagnostics data are scanned using system blocks. The system function SFC 13 DPNRM_DG is available for this with conventional DP standard slaves. The SFC 59 RD_REC is used with DP S7 slaves in order to read the data record DS1 with the diagnostics data. DPV1 slaves are able to provide more comprehensive diagnostics data which can be read with the system function block SFB 52 RDREC. The modules are addressed by means of the logical module starting address of the user data or by means of the diagnostics addresses.

Transfer memory on intelligent DP slaves

In the case of compact and modular DP slaves, the addresses of the inputs and outputs are located together with the addresses for the central modules in the address volume of the master CPU. In the case of intelligent slaves, the master CPU has no direct access to the input/output modules of the DP slave. Every intelligent DP slave therefore has a transfer memory whose size depends on the CPU used. The transfer memory can be divided into several areas of different length and data consistency. The individual areas then respond like modules whose lowest address is the module starting address. From the viewpoint of the master CPU, the intelligent DP slave then appears as a compact or modular DP slave, depending on the division (Figure 20.10).

When configuring the slave, you can configure the individual areas of the transfer memory as inputs or outputs with the “module starting address” and the area length. Exception: if the CP 342-5DP provides the DP interface for the intelligent slave, the division of its transfer memory is only configured when coupling to the DP master system. The addresses of the transfer memory must not overlap with those of the central modules in the intelligent DP slave. If the addresses are present in the process image, the areas can be handled by the user program like inputs and outputs, otherwise like peripheral inputs and outputs. If the slave CPU possesses partial process images, you can assign a partial process image to each area.



- ① Each bus interface has a diagnostics address which is set in the object properties of the interface in the “Addresses” tab (double click “DP” line in the master CPU).
- ② The station and ist bus interface have a diagnostics address which is set in the object properties of the station in the “General” tab (double click the DP slave station in the hardware configuration of the DP master).
- ③ Each bus interface has a diagnostics address which is set in the object properties of the interface in the “Addresses” tab (double click the “DP” line in the slave CPU).
- ④ The station and ist bus interface have a diagnostics address which is set in the object properties of the station in the “Mode” tab (double click the “DP” line in the slave CPU).

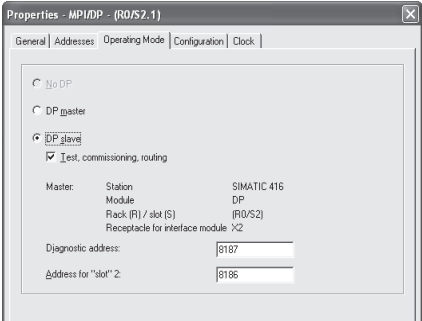
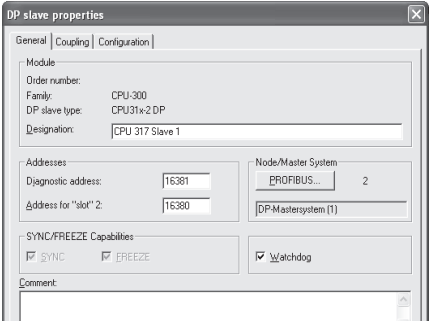
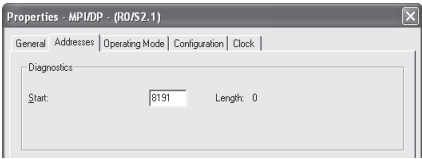
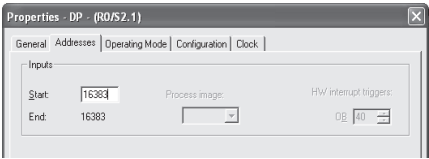


Figure 20.9 Diagnostics Addresses in a DP Master System

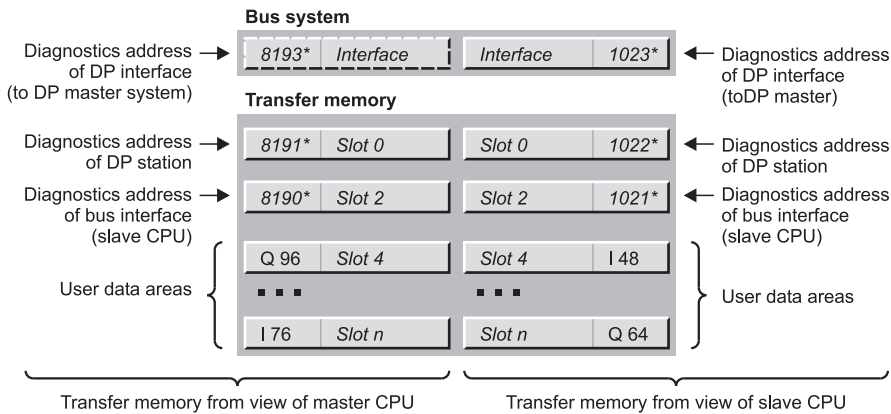


Figure 20.10 Transfer Memory on Intelligent DP Slaves

When coupling to the DP master system, you supplement the configuration at the master end by the “module starting addresses” from the viewpoint of the master CPU and by the transmission direction. You assign inputs on the slave side to outputs on the master side and vice versa. If the addresses are within the process image, the areas can be handled by the user program like inputs and outputs, otherwise like peripheral inputs and outputs. If the master CPU possesses process image partitions, you can also assign a process image partition to each area. From the viewpoint of the master CPU, the addresses of the transfer memory must not overlap with addresses of other modules in the (centralized) S7 station, neither with the addresses of the central modules nor with the addresses in other DP master systems assigned to the master CPU.

You also define the diagnostics addresses from the viewpoint of the intelligent DP slaves during configuration of these. You determine the diagnostics addresses from the viewpoint of the master CPU during coupling of the intelligent DP slave to the DP master system.

20.4.2 Configuring PROFIBUS DP

General procedure

You configure the distributed I/O on the PROFIBUS DP in essentially the same way as the centralized modules. Instead of arranging modules in a mounting rack, you assign DP stations

(PROFIBUS nodes) to a DP master system. The following order is recommended for the necessary actions:

- 1) Create a new project or open an existing one with the SIMATIC Manager.
- 2) Create a PROFIBUS subnetwork in the project with the SIMATIC Manager and, if required, set the bus profile.
- 3) Use the SIMATIC Manager to create the master station in the project that is to accommodate the DP master, e.g. an S7-400 station.

If your system contains intelligent DP slaves, you also create the relevant slave stations at this point, e.g. S7-300 stations.

You start the Hardware Configuration by opening the master station.

- 4) With the Hardware Configuration, you place a DP master in the master station. This can be, for example, a CPU with integral DP interface. You assign the previously created PROFIBUS subnetwork to the DP interface and you then have a DP master system. You also define the DP mode in the “Mode” tab: DPV1 or S7-compatible. You can also configure the remaining modules later. Save and compile the station.
- 5) If you have created an S7 station for an intelligent DP slave, you open this in the Hardware Configuration and you “plug in” the module with the desired DP interface,

e.g. an S7-300 CPU with integral DP interface or an ET200pro basic module IM 154-8/CPU. If you set the DP interface as “DP slave”, assign the previously created PROFIBUS subnetwork to the DP interface and configure the transfer memory from the viewpoint of the DP slave.. You can also configure the remaining modules later. Save and compile the station.

Proceed in the same way for the remaining stations intended for intelligent DP slaves.

- 6) Open the master station with the DP master system and use the mouse to drag the PROFIBUS nodes (compact and modular DP slaves) from the hardware catalog to the DP master system. Assign node addresses and, if necessary, set the module starting address and the diagnostics address.
- 7) If you have created intelligent DP slaves, drag the relevant icon (in the hardware catalog under “PROFIBUS DP” and “Already configured stations”) with the mouse to the DP master system.

Open the icon and assign the already configured DP slave (“Connect”), assign a node address and configure the user data interface from the viewpoint of the DP master (or from the viewpoint of the central master CPU). Proceed in the same way with every intelligent DP slave.

- 8) Save and compile all stations. The DP master system is now configured. You can now supplement the configuration with centralized modules or with further DP slaves.

You can also represent the DP master system configured in this way graphically with the Network Configuration tool. Open Network Configuration by, for example, double-clicking on a subnetwork. Select VIEW → DP SLAVES/IO DEVICES to display the slaves. You can also create a DP master system (or more precisely, assign the nodes to a PROFIBUS subnetwork) with the Network Configuration tool. You parameterize the stations after opening them with the Hardware Configuration. Here too, you must first set up an intelligent DP slave before you can integrate it into a DP master system.

Configuring the DP master

You must have created a project and an S7 station with the SIMATIC Manager. Open the S7 station and position the DP master module. This can be an S7 or ET200 CPU with DP interface, a CP 342 communications processor (with S7-300), or an IM 467 interface module (with S7-400) (see Chapter 2.3, “Configuring Stations”).

Now drag the DP master module from the Hardware Catalog to the configuration table of the mounting rack. You may already have selected a CPU with DP connection. In the line below, the DP master is displayed with a connection to a DP master system in the station window (broken black-and-white bar).

When placing the DP master module, you select in a window the PROFIBUS subnetwork to which the DP master system is to be assigned and the node address to be assigned to the DP master. You can also create a new PROFIBUS subnetwork in this window.

In the “Mode” tab you also define the DP mode with which the DP master is to work. This mode applies to the complete DP master system.

Together with the DP master, the hardware configuration displays a DP master system in the station window (black/white dashed rail). If there is no DP master system available (it may be that it is obscured behind an object or is outside the visible area), create one by selecting the DP master in the configuration table, and then selecting INSERT → MASTER SYSTEM. If there is no DP master system available (it may be that it is obscured behind an object or it is outside the visible area), create one by selecting the DP master in the configuration window and then selecting INSERT → MASTER SYSTEM. You can change the node address and the connection to the PROFIBUS subnetwork by selecting the module and then making your changes with the “Properties” button on the “General” tab under EDIT → OBJECT PROPERTIES.

CP 342-5DP as DP master

If a CP 342-5DP is the DP master, place it in the configuration table of the station, select it and

then EDIT → OBJECT PROPERTIES. Set “DP Master” on the “Mode” tab.

The “Addresses” tab shows the user data address occupied by the CP in the address area of the CPU. From the viewpoint of the master CPU, the CP 342-5DP is an “analog module” with a module starting address and 16 bytes of user data.

Only DP standard slaves, or DP S7 slaves that behave like DP standard slaves, can be connected to a CP 342-5DP as DP master. You can find the suitable DP slaves in the hardware catalog under “PROFIBUS DP” and “CP 342-5DP as DP master”. Select the desired slave type and drag it to the DP master system.

The transfer memory as DP master has a maximum length of 240 bytes. It is transferred as one with the loadable blocks FC 1 DP_SEND and FC 2 DP_RECV (included in the *Standard Library* under the *Communication Blocks* program).

The data consistency covers the entire transfer memory.

You read the diagnostics data of the connected DP slaves with FC 3 DP_DIAG (e.g. station list, diagnostics data of a specific station). FC 4 DP_CTRL transfers control jobs to the CP 342-5DP (e.g. SYNC/FREEZE command, CLEAR command, set operating state of the CP 342-5DP).

If you select CPU or CP 342-5DP, VIEW → ADDRESS OVERVIEW shows you a list of the assigned addresses, inputs and/or outputs. You can also screen the existing address gaps.

Configuring compact DP slaves

The compact DP slaves are to be found in the hardware catalog under “PROFIBUS DP” and the relevant sub-catalog, e.g. ET200B. Click on the DP slave selected and drag it to the icon for the DP master system.

You will see the properties sheet of the station; here, you set the node address and any diagnostics address. Then the DP slave appears as an icon in the upper section of the station window and the lower section contains a configuration table for this station.

A double-click on the icon in the upper section of the station window opens a dialog box with

one or more tabs in which you set the desired station properties. In the lower sub-window, you then see the input/output addresses. Double-clicking on an address line shows you a window where you can change the suggested addresses.

The lower sub-window shows optionally the configuration table of the selected DP slave or of the master system (toggle with the “arrow” button).

Configuring modular DP slaves

The modular DP slaves can be found in the hardware catalog under “PROFIBUS DP” and the relevant sub-catalog, e.g. ET200M.

Click on the selected interface module (basic module) and drag it to the icon for the DP master system. This screens the properties sheet for the station; here, you set the node address and any diagnostics address. Then the DP slave appears as an icon in the upper section of the station window and the lower section contains configuration table for this station.

Now place the modules that you can find in the hardware catalog *under the selected interface module (!)* in the configuration table. Double-clicking on the line opens the properties sheet of the module and allows you to parameterize the module.

The lower sub-window shows either the configuration table of the selected DP slave or of the master system (toggle with the “arrow” button).

Combining modules

Digital electronic modules with two or four bit channels, e.g. ET 200S or ET 200pro, initially occupy one address each with one byte in the configuration table. After all modules have been configured, you can use the “Pack addresses” button in the configuration table to remove the gaps between the bit channels; fewer addresses are then occupied. The address areas for inputs, outputs, and motor starters are packed separately.

Please note the following special features of a “packed” module:

- ▷ Slot assignment is no longer possible; the CPU cannot determine a geographical address for this module.

- ▷ No module status information for this module can be read.
- ▷ Interrupts cannot be assigned to a “packed” address. For this reason, a diagnostic address is assigned to a module (indicated by italics and a star in the configuration table). You can receive interrupt information under this address.
- ▷ “Pack addresses” and “Insert/remove module interrupt” are mutually exclusive.

General handling of options

With the handling of options you prepare an ET 200S or ET 200pro station for a future expansion. This means you can start operation with a small configuration and subsequently upgrade the station to a (previously) planned maximum configuration without modification of the hardware configuration. To do this, you require appropriately equipped IM interface modules and PM power modules. The user program in an ET 200S can scan and control the slot assignment during runtime via a control and feedback interface.

Please note with runtime calculations, as required e.g. with isochronous mode, that the maximum planned configuration must be applied.

Depending on the version of the interface module, you can use the handling of options with or without reserve modules. With the handling of options without reserve modules, you configure the envisaged maximum configuration of the ET 200 station, where you position the modules to be inserted later in the rear slots. Before the station commences with cyclic data exchange with I/O access operations following the initial startup, you use the user program to define the slots identified as “reserve” via the control interface, and then enable operation. Following replacement by the planned module, you cancel the identification as “reserve” using the user program.

With the handling of options with reserve modules, you can carry out the upgrading at any slots, but assignment of the slots without gaps must be guaranteed. You configure the planned maximum configuration and in the hardware configuration you identify the slots which, for the time being, are assigned with reserve mod-

ules. The reserve modules report the signal state “0” as the substitute value for digital inputs, and the value W#16#7FFF for analog channels. Following replacement by the planned module, you cancel the identification as “reserve” using the user program via the control interface. The use of reserve modules presents the advantage that the wiring for the maximum configuration can be completely provided right at the beginning.

Handling of options with ET 200S

Prerequisite: You have at least configured the ET 200S station with the power module. You activate the handling of options in both the IM interface module and the PM power module.

To activate the handling of options in the interface module, open the station's Properties window (select station and EDIT → OBJECT PROPERTIES). On the “Option handling” tab, switch on the handling of options using the “Option handling active” check box. With a correspondingly designed module, you can choose between “Without reserve module” and “With reserve module”. When selecting “Without reserve module”, terminate configuration using “OK”.

If you have chosen “With reserve module” or if this selection is missing for the interface module, now select the slots in the “Parameters” window which are only to be equipped later with the envisaged modules. The reserve modules are not configured; you configure the station with the planned maximum configuration of electronic modules envisaged for later.

Open the Properties window of the power module (select module in the configuration table and EDIT → OBJECT PROPERTIES).

On the “Addresses” tab, check the “Option handling” check box. In the address area of the station, an additional eight bytes of digital inputs are then assigned for the feedback interface and an additional eight bytes of digital outputs for the control interface. The user program can query via the feedback interface whether the handling of options is active and whether the envisaged (configured) module is inserted in a slot and ready for operation.

If you activate the handling of options immediately following “insertion” of the power mod-

ule and before configuration of the electronic modules, the addresses of these interfaces are set as standard to the beginning of the ET 200S address area.

Handling of options with ET 200pro

Prerequisite: You have at least configured the ET 200pro station with the power module. The power module must be designed for the handling of options (PM-O). You activate the handling of options in the IM interface module.

To activate the handling of options in the interface module, open the station's Properties window (select station and EDIT → OBJECT PROPERTIES). On the "Operating parameters" tab, check the "Option handling" checkbox.

Configuring a CPU with integral DP interface as an intelligent DP slave

With an appropriately equipped CPU, you can parameterize the station either as a DP master

station or as a DP slave station. Before the station can be connected as a DP slave to a DP master system, it must be created. The procedure for doing this is exactly the same as that for a "normal" station; insert an S7 station into the project using the SIMATIC Manager and open the *Hardware* object. Drag a mounting rack to the window in the Hardware Configuration and place the desired modules. For configuring the DP slave, it is enough to place the CPU; you can add all other modules later.

When inserting the CPU, the properties sheet of the PROFIBUS interface is screened. Here, you must assign a subnetwork to the DP interface and you must assign an address. If the PROFIBUS subnetwork does not yet exist in the project, you can create a new one with the "New" button. This is the subnetwork to which the intelligent slave will later be connected.

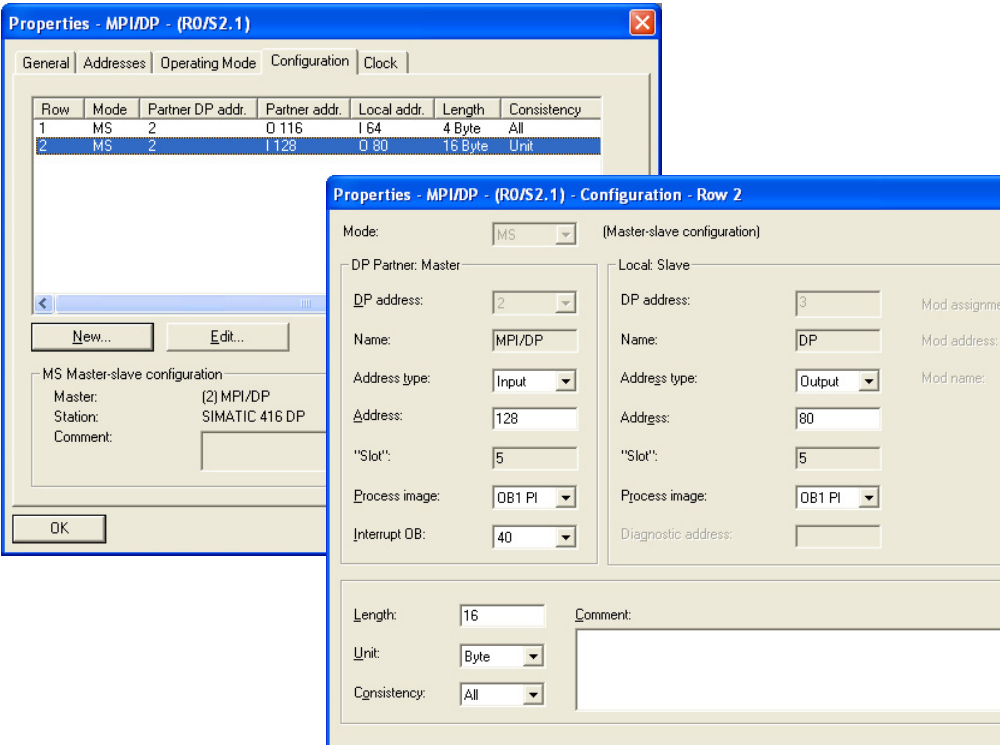


Figure 20.11 Configuring the Transfer Memory of an Intelligent Slave with Integral DP Interface

You can open the properties sheet of the interface by selecting the DP interface and then EDIT → OBJECT PROPERTIES or by double-clicking on the interface. On the “Mode” tab, select the option “DP Slave”. Now you can configure the user data interface on the “Configuration” tab from the viewpoint of the DP slave. Select MS (master/slave configuration) as the mode, and define the structure and addresses of the transfer interface from the viewpoint of the slave CPU.. Chapter 20.4.1, “Addressing PROFIBUS DP” provides information on the user data interface under “Transfer memory on intelligent DP slaves”.

If the intelligent DP slave is already coupled to a DP master system, you can also immediately enter the user data addresses from the viewpoint of the DP master (Figure 20.11).

The size and structure of the transfer memory is CPU-specific. On the CPU 315-2DP, for example, you can divide the entire transfer memory into up to 32 address areas that you can access separately. Such an address area can be up to 32 bytes in size. The entire transfer memory can have up to 244 input addresses and 244 output addresses.

The addresses defined here are located in the address volume of the slave CPU. These addresses must not overlap with addresses of the central or distributed modules in the DP slave station. The lowest address of an address area is the “module starting address”.

The user program in the slave CPU gets diagnostics information from the DP master via the diagnostics addresses specified on this tab.

You terminate configuration of the intelligent DP slave with STATION → SAVE AND COMPILE. Connecting the intelligent DP slave into the DP master system is described below.

Configuring an ET200pro as an intelligent DP slave

Configuring an ET200pro station is very similar to configuring an S7-300 station. Insert a SIMATIC 300 station in the SIMATIC Manager under the project and open the *Hardware* object.

Now drag an interface module with CPU functionality *IM xxx CPU* under “PROFIBUS DP”

and “ET 200pro” in the hardware catalog to the free window or select it by double-clicking on it. On the displayed properties sheet of the Ethernet interface, set “not networked”.

EDIT → OBJECT PROPERTIES in the case of a selected interface X1, or double-click the MPI/DP interface to open the Properties window. On the “General” tab, select PROFIBUS as the interface, and in the Properties window of the PROFIBUS interface select the station address and the PROFIBUS subnet. If the PROFIBUS subnet does not yet exist in the project, you can create one with the “New” button. This is the subnet to which the intelligent slave will later be connected.

On the “Operating Mode” tab, select the option “DP Slave”. The meanings of the addresses on this tab and of the address on the “Addresses” tab are described in Chapter 20.4.1, “Addressing PROFIBUS DP” under , “Diagnostics address”.

Now you can configure the user data interface on the “Configuration” tab from the viewpoint of the DP slave. Select MS (master/slave configuration) as the mode, and define the structure and addresses of the transfer interface from the viewpoint of the ET200pro CPU. If the intelligent DP slave is already coupled to a DP master system, you can also immediately enter the user data addresses from the viewpoint of the DP master (Figure 20.11). You can find information on the user data interface in Chapter 20.4.1, “Addressing PROFIBUS DP” under, “Transfer memory on intelligent DP slaves”.

On the IM 154-8 CPU interface module, you can divide the entire transfer memory into up to 32 address areas that you can address separately. Such an address area can be up to 32 bytes in size. The entire transfer memory can encompass up to 244 input addresses and 244 output addresses.

The locally defined addresses are within the address volume of the ET200pro CPU. These addresses must not overlap addresses of the central or distributed modules in the ET200pro station. The lowest address of an address area is the “module starting address”.

Further configuration of the ET200pro station is carried out in the same way as that for an S7-300 station with fixed slot addressing. You can

only arrange modules that can be found in the hardware catalog under the interface module used.

You terminate configuration of the intelligent DP slave with **STATION → SAVE AND COMPILE**. Connecting the intelligent DP slave into the DP master system is described below.

Configuring an IM 151/CPU as an intelligent DP slave

If you want to generate an ET200S as an intelligent DP slave, first insert a SIMATIC 300 station under the project in the SIMATIC Manager and open the *Hardware* object.

Now drag an interface module with CPU functionality *IM xxx CPU* under "PROFIBUS DP" and "ET 200S" in the hardware catalog to the free window or select it by double-clicking on it.

You will see a configuration table like the one for a SIMATIC 300 station. The intelligent IM 151 of the ET200S station is present here instead of the CPU.

Double-clicking on the IM 151 line opens the window for the IM properties; double-clicking on the DP interface opens the properties window of the interface. If you have not already done so, set the interface type "PROFIBUS", the node address, and the PROFIBUS subnet on the "General" tab.

Set the address areas for the transfer memory from the viewpoint of the DP slave. If the intelligent DP slave is already coupled to a DP master system, you can also immediately enter the user data addresses from the viewpoint of the DP master (Figure 20.11). The maximum size of the user data area is 32 bytes of inputs and 32 bytes of outputs for the IM 151/CPU interface module. You can divide this area into eight sub-areas with different data consistencies. The slave program obtains diagnostic information from the DP master via the diagnostic address.

Further configuration of the ET200S station is carried out in the same way as for an S7-300 station with fixed slot addressing. You can only arrange the modules listed in the Hardware Catalog under "IM151/CPU".

Select **STATION → SAVE AND COMPILE** to conclude configuring the intelligent DP slave. Inte-

gration of the intelligent DP slave into the DP master system is described further below.

Configuring an S7-300 station with CP 342-5DP as an intelligent slave

If you insert an S7-300 station in the SIMATIC Manager, open the *Hardware* object and configure a "normal" S7-300 station. Among other things, you arrange a CP 342-5DP communications processor in the configuration table.

When inserting the station, the properties sheet of the DP interface appears; the subnetwork to which the intelligent DP slave is later to be connected is to be assigned to the DP interface here and you must also assign the node address.

To open the properties window, select the CP 342-5DP and then **EDIT → OBJECT PROPERTIES**, or double-click on the CP 342-5DP. On the "Mode" tab, select the option "DP Slave".

The "Addresses" tab shows the transfer memory from the viewpoint of the slave CPU. The maximum size of the transfer memory with the CP 342-5DP as DP slave is 240 bytes each for inputs and outputs, which you can divide into a maximum of 63 address areas following coupling to the master system.

STATION → SAVE AND COMPILE terminates configuration of the intelligent DP slave.

Connecting an intelligent DP slave to a DP master

You must have created a project and configured a DP master station and the intelligent DP slave (in each case at least with the DP interface). The DP master and the DP slave must be configured for the same PROFIBUS subnetwork.

Open the master station; a DP master station (black/white dashed rail) must be present, otherwise generate it with the DP interface selected using **INSERT → MASTER SYSTEM**.

In the hardware catalog under "PROFIBUS DP" and "Configured stations", you can find the objects which represent the intelligence slaves: "CPU31x" and "CPU41x" stand for S7-300 or S7-400 stations with an integrated DP slave, "ET200pro/CPU" and "ET200S/CPU" stand for stations configured as a DP slave, and in the folder "S7-300 CP342-5 DP"

you can find the proxies for S7-300 stations with CP 342-5 as DP slave interface. Select the desired slave type and drag it to the DP master system.

CPU, ET200pro or ET200S as DP slave

Dragging to a DP master system or double-clicking on the DP slave opens the properties sheet. The DP slaves already configured for this PROFIBUS subnetwork are listed on the "Connection" tab. Select the desired DP slave and click on the "Connect" button. This causes the active connection to be carried out at the bottom of the same dialog box.

On the "General" tab, you set the diagnostics address of the DP slave from the viewpoint of the master station.

On the "Configuration" tab, you now set the addresses of the user data interface from the viewpoint of the DP master. The output addresses on the master are the input addresses of the slave and vice versa. Chapter 20.4.1, "Addressing PROFIBUS DP" contains more information on the user data interface under "Transfer memory on intelligent DP slaves".

CP 342-5DP as DP slave

You open the properties sheet by dragging to the DP master system or by double-clicking on the DP slave. The DP slaves already configured for this PROFIBUS subnetwork are listed on the "Connection" tab. Select the desired DP slave and click on the "Connect" button. The active connection is then shown further down in the same dialog box.

When the DP slave is selected, its configuration table is shown in the lower section of the station window. You now configure the transfer memory: Drag the modules with the required properties from the selection under the used CP, or the universal module, into the configuration table.

EDIT → OBJECT PROPERTIES with the module selected in the bottom part of the window, or a double-click on the table line, opens a window in which you can set the start address. In the properties of the universal module, you can set whether an empty location, an input or output area, or both is to be displayed.

If a CP 342-5DP is the DP master, structuring of the transfer memory can be omitted because the CP 342-5DP transfers the entire transfer area in one piece.

When dividing the transfer memory, you arrange the address areas together without gaps starting from byte 0. You access the entire assigned transfer memory in the slave CPU with the loadable blocks FC 1 DP_SEND and FC 2 DP_RECV (included in the *Standard Library* under the *Communication Blocks* program).

The data consistency covers the entire transfer memory.

On the "General" tab, you set the diagnostics address of the DP slave from the viewpoint of the master station. The diagnostics data are read with FC 3 DP_DIAG (in the master station).

Chapter 20.4.1, "Addressing PROFIBUS DP" contains more information on the user data interface under "Transfer memory on intelligent DP slaves".

Configuring the DP/DP coupler

The DP/DP coupler connects two PROFIBUS subnets. It is configured as a modular DP slave in each of the two subnets.

Prerequisite: Both subnets, each with a DP master system, must be configured. Open one of the stations with the DP master. In the hardware catalog under "PROFIBUS DP" and "Network components", you will find the *DP/DP coupler, Release 2* that you can drag to the DP master system with the mouse.

The properties sheet of the PROFIBUS interface appears on which you set the node address. You set the diagnostic address and further parameters on the "General" and "Parameterize" tabs in the object properties of the DP/DP coupler.

With a selected DP/DP coupler, the configuration table for the transfer memory appears. Now "Connect" the desired modules listed in the hardware catalog under DP/DP coupler into the configuration table, without gaps and starting at slot 1. You can set the universal module to the desired number of inputs and outputs. The user data addresses that you specify in the

module properties are in the address area of the opened DP master CPU.

Configure the second part of the DP/DP coupler in the same way. The structure of the transfer memory must agree with that of the first part. Please note that inputs on one side are outputs on the other side, and vice versa. The addresses in both parts of the DP/DP coupler are oriented to the address assignments of the relevant master CPU and can differ from each other.

Configuring the DP/AS-i link

You configure the DP/AS-Interface link like a modular DP slave. You will find the modules that you can drag to the DP master system, such as the DP/AS-i link 20 below, in the hardware catalog under “PROFIBUS DP” and “DP/AS-i”. Then you set the properties of the DP section and configure the AS-i slaves –

depending on the link on the properties sheet or in the configuration table.

The AS-i master system with the AS-i slaves is not displayed as a subnet by the hardware configuration.

DP/AS-i-Link 20E and DP/AS-i-Link Advanced

“Drag” the DP/AS-i Link 20 from the hardware catalog to the bar of the PROFIBUS DP master system. You can set the node address in the displayed window or in the properties of the DP slave under the “PROFIBUS...” button.

A configuration table with the AS-i interfaces and AS-i slaves appears. If you have selected the double master for *DP/AS-i-Link Advanced*, the configuration table is designed for both masters.

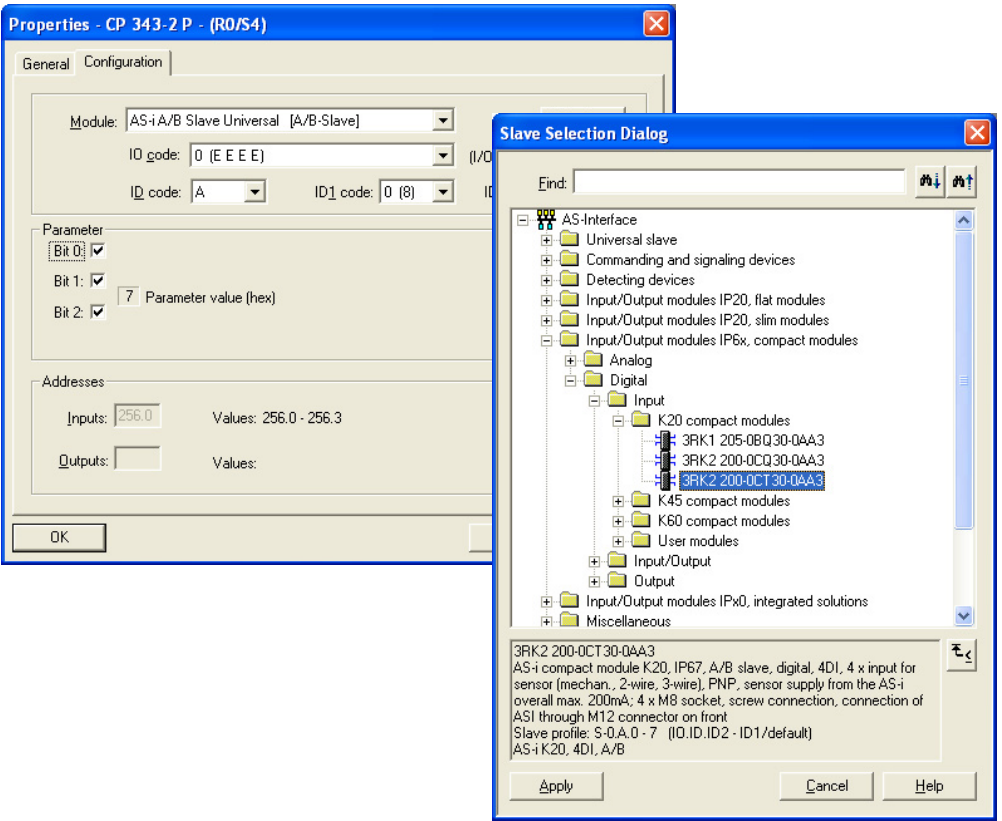


Figure 20.12 Properties and selection dialog for AS-i slaves in the case of DP/AS-i Link Advanced

Now set the address area under which you want to access the AS-i slaves from the user program. EDIT → OBJECT PROPERTIES with a selected AS-i master interface (in the configuration table), or double-click on the master line to open the properties dialog. On the "Digital Addresses" tab, you can set the start address and the inputs; this address also applies to the outputs. You can select different values for the reserved area lengths for inputs and outputs. This tab also contains the "Pack" and "Sort" buttons with which you can optimize the address assignments after you have configured the AS-i slaves.

Now "drag" the placeholder for an AS-i slave positioned under the link from the hardware catalog to the configuration table. Repeat this procedure for all scheduled AS-i slaves. EDIT → OBJECT PROPERTIES with a selected AS-i slave (in the configuration table), or double-click on the slave line, opens the properties dialog. Set the slave properties on the "Configuration" tab. By using the "Selection" button, you are provided with all the AS-i slaves known to the hardware configuration (Figure 20.12).

Configuring the DP/RS232C link

You configure the DP/AS-Interface link as with a modular DP slave. You can find the DP/RS232C link in the hardware catalog under "PROFIBUS DP", "Other field devices" and "Gateway" and can drag it to the DP master system.

The properties sheet of the PROFIBUS interface appears on which you set the node address. You set the diagnostic address and further parameters on the "General" and "Parameterize" tabs in the object properties of the DP/RS232C link.

With the DP/RS232C link selected, you are provided with the configuration table. Now "Connect" the desired modules listed in the hardware catalog under DP/RS232C link into the configuration table, without gaps and starting at slot 1. You can set the universal module to the desired number of inputs and outputs. The user data addresses that you specify in the

module properties are in the address area of the opened DP master CPU.

20.4.3 Special Functions for PROFIBUS DP

GSD files

You can "post install" DP slaves that are not included in the module catalog. For this purpose, you require the type file tailored to the slave (GSD file, General Station Description, device database file). From GSD version 3 onwards, DP slaves installed with a GSD file support the DPV1 functionality. Select OPTIONS → INSTALL GSD FILE in the Hardware Configuration and specify the directory of the GSD file or another STEP 7 project in the window that appears. STEP 7 accepts the GSD file and displays the slave in the hardware catalog under "PROFIBUS DP" and "Additional Field Devices".

STEP 7 saves the GSD files in the directory ...\\Step7\\S7DATA\\GSD. The GSD files deleted during subsequent installation or importing are saved in the subdirectory ...\\GSD\\BKPn. From here, they can be restored with OPTIONS → INSTALL GSD FILE.

Configuring SYNC/FREEZE groups

The SYNC control command causes the DP slaves combined as a group to output their output states simultaneously (synchronously). The FREEZE control command causes the DP slaves combined as a group to "freeze" the current input signal states simultaneously (synchronously), in order to allow them to be then fetched cyclically by the DP master. The UNSYNC and UNFREEZE control commands revoke the effect of SYNC and FREEZE respectively.

It is a requirement that the DP master and the DP slaves have the relevant functionality. From the object properties of a slave, you can see which command it supports (select DP slave, EDIT → Object Properties, "General" tab under "SYNC/FREEZE capability").

Per DP master system, you can form up to 8 SYNC/FREEZE groups that are to execute either the SYNC command or the FREEZE

command or both. You can assign any DP slave to a group; on the CP 342-5DP of a specific version, one DP slave can be represented in up to 8 groups.

When you call SFC 11 DPSYC_FR, you cause the user program to output a command to a group (see Chapter 20.4.7, “System blocks for distributed I/O”). The DP master then sends the relevant command simultaneously to all DP slaves in the specified group.

You configure the SYNC/FREEZE groups following configuration of the DP master system (all DP slaves must be present in the DP master system). Select the DP master system (broken black-and-white bar) and select EDIT → OBJECT PROPERTIES. In the window that then

appears, you first establish on the “Group Properties” tab the group commands to be executed (Figure 20.13), then you assign the DP slaves to the individual groups on the “Group Assignment” tab.

Here you select each of the DP slaves listed with its node number one after the other and select in each case the group to which it is to belong. If a DP slave cannot execute a specific command, e.g. FREEZE, it cannot be assigned to a group which contains this command. Terminate configuring of the SYNC/FREEZE groups with OK.

Please note that when configuring bus cycles of the same length (equidistant), groups 7 and 8 acquire a special meaning.

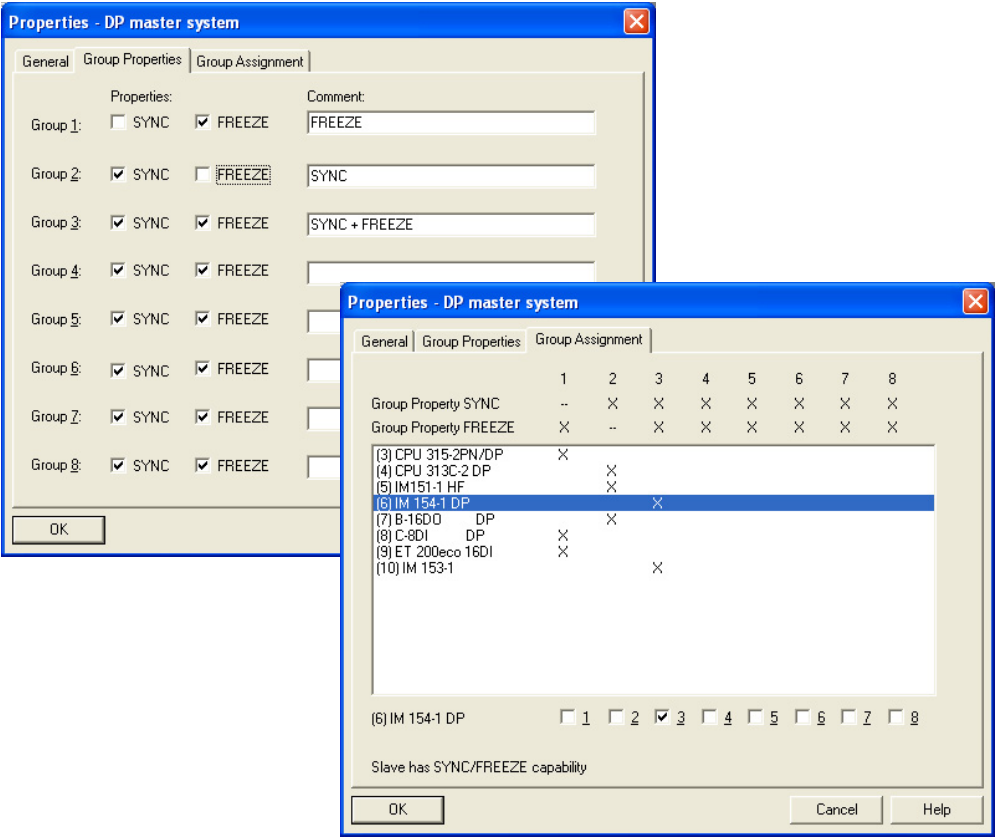


Figure 20.13 Configuring of the SYNC/FREEZE groups

Configuring PROFIBUS PA

For configuring a PROFIBUS PA master system and for parameterizing the PA field devices, use the Hardware Configuration tool (STEP 7 V5.1 SP3 and later) and the SIMATIC PDM optional software (with an earlier version of STEP 7). With the Hardware Configuration, you build up the connection to the DP master system with the DP/PA link: in the Hardware Catalog under “PROFIBUS DP” and “DP/PA Link”, drag the IM 157 interface module to the DP master system. With the DP slave, a PA master system is simultaneously created in its own PROFIBUS subnetwork (45.45 kbits/s); it is indicated with a broken black-and-white bar.

The DP/PA coupler transfers the data unmodified and uninterpreted between the bus systems; for this reason, it is not parameterized. The PA field devices are addressed by the DP master. They can be incorporated as DP standard slaves into the Hardware Configuration of STEP 7 via a GSD file. Following this, you will find the PA field devices in the hardware catalog under “PROFIBUS DP” and “Other Field Devices”.

Configuring direct data exchange (slave-to-slave communication)

In a DP master system, the DP master has exclusive control over the slaves assigned to it. With appropriately equipped stations, another node (master or intelligent slave, referred to as the “receiver” or “subscriber”) can monitor the PROFIBUS subnetwork to learn which input data a DP slave (“sender” or “publisher”) is sending to “its” master. This direct data exchange is also called “lateral communication”. In principle all DP slaves from a specific revision level can function as senders in direct data exchange.

You configure direct data exchange with the Hardware Configuration in the Properties window of the DP slave (receiver) when all the stations on the PROFIBUS subnetwork are connected. Open the receiver station and select the DP interface and then EDIT → OBJECT PROPERTIES. The “Configuration” tab contains the transfer interface between the DP slave and the DP master. Click the “New” button, and set DX mode (direct data exchange) in the configura-

tion window which is displayed. Also define the parameters for the DP partner (sender) in the same window.

You can also use direct data exchange between two DP master systems on the same PROFIBUS subnetwork. For example, the master in master system 1 can monitor the data of a slave in master system 2 in this way.

Configuring constant bus cycle times and isochrone mode

Equidistance

Normally, the DP master controls the DP slaves assigned to it cyclically without a pause. With S7 Communication, such as when the programming device executes modify functions via the PROFIBUS subnetwork, this can result in variations in the time intervals. If, for example, the outputs are to be modified via DP slaves at a regular interval, you can set constant bus cycles with the appropriately equipped DP master. For this purpose, the DP master must be the only Class 1 master on the PROFIBUS subnetwork. Constant bus cycle time behavior is possible with the bus profiles “DP” and “User-Defined”. The PROFIBUS subnet must not be project-wide and no H system or CiR (Configuration in Run) object may be connected.

If you configure SYNC/FREEZE groups in addition to the equidistant behavior, please observe the following:

- ▷ For DP slaves in group 7, the DP master automatically initiates a SYNC/FREEZE command in every bus cycle. Initiation per user program is prevented.
- ▷ Group 8 is used for the constant bus cycle time signal and is disabled for DP slaves. You cannot configure constant bus cycle time if you have already configured slaves for group 8.

Isochronous mode

Reference is made to isochrone mode if a program is executed in synchronism with the PROFIBUS DP cycle. In conjunction with constant bus cycle times, this results in reproducible response times of equal length to the I/O that include distributed signal acquisition, signal transfer via PROFIBUS and program execution

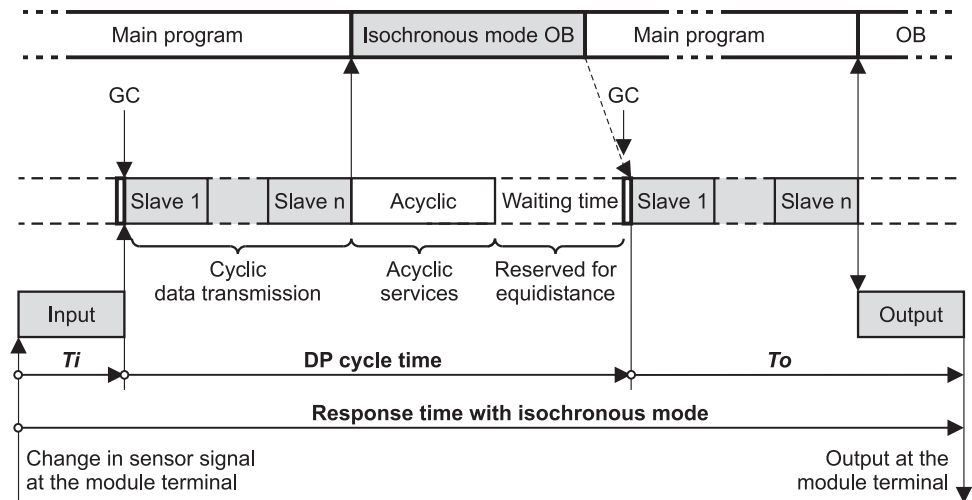


Figure 20.14 Response time in the case of isochronous mode and equidistance time

including process image updating. The user program executed in isochrone mode is present in the organization blocks OB 61 to OB 64. The system functions SFC 126 SYNC_PI and SFC 127 SYNC_PO are available for updating of the process image in isochronous mode (see Chapter 21.8, “Synchronous Cycle Interrupts”).

Figure 20.14 shows the timers involved in isochronous mode. T_i is the time required for reading in the process values. It contains the execution time in the input modules or electronics modules, and, in the case of modular DP slaves, the transfer time on the backplane bus. At the end of T_i , the input information for transfer using the global control command is available. Then the equidistance time begins. It is the time between two global control commands and encompasses the transfer to the subnet as well as the execution of the isochronous interrupt OB. Between completion of the execution of this OB to the next global control command there must be time for execution of the main program.

T_o is the time required for outputting the process values. It begins with the global control command and comprises the transfer time on the subnet as well as the processing time in the output modules or electronics modules. In the case of modular DP slaves, the transfer time on the backplane bus is also added. The response

time in the case of isochronous mode is the total of the times T_i , equidistance time, and T_o .

Correspondingly designed DP slaves allow a reduction in the response time thanks to *overlapping isochronous mode*. This involves overlapping updating of the input and output signals (overlapping of T_i and T_o). In this case, you must enter the individual times for the participating modules. If isochronous modules have both inputs and outputs, overlapping of T_i and T_o is not possible.

Configuring the isochronous mode

A prerequisite for configuration of isochronous mode is the constant bus cycle time and the corresponding functionality of the participating DP components.

Add a station and a CPU module with integrated DP interface in the project, for example an S7-300 station with a CPU 317-2PN/DP. To insert a DP master system, set the type "PROFIBUS" on the "General" tab in the interface properties of the MPI/DP interface, and the option "DP Master" on the "Mode" tab. Click the "Properties..." button on the "General" tab, and connect the interface to a PROFIBUS subnet.

Click the "Properties..." button to activate the constant bus cycle times. Select the "Network Settings" tab in the properties window of the

PROFIBUS interface. Note that the constant bus cycle times can only be set with the bus profiles “DP” and “User-defined”. Click the “Options” button and the “Activate constant bus cycle times” checkbox in the options window which is then displayed.

For isochronous mode, specify additionally the times T_i and T_o on this tab. Either select the “Times T_i and T_o same for all slaves”, or set the times individually in the slave properties.

Each *module* or each *submodule* participating in isochronous mode must have a user data address in a process image partition that is updated in isochronous mode by the system functions SFC 126 SNYC_PI and SFC 127 SNYC_PO. You make the assignment between the user data addresses and a process image partition in the module or submodule properties when setting the address.

The isochronous modules must be made known to the *DP interface module*. In the properties

window of the DP slave, activate the option “Synchronize DP slave to constant bus cycle time DP cycle” on the “Isochrone mode” tab. Here you also select the modules or submodules involved in isochronous mode.

To update T_i and T_o , click in the properties of the DP master system on the “Properties” button on the “General” tab. In the displayed window, change to the “Network Settings” tab and click on the “Options” button. When the “Recalculate” button is activated, STEP 7 updates all times involved in isochronous mode. You can modify the suggested equidistance time but not below the displayed minimum time. The “Details” button shows the individual proportions of the equidistance time. Please note that the equidistance time increases the more programming devices are connected directly to the PROFIBUS subnet and the more intelligent DP slaves are in the DP master system.

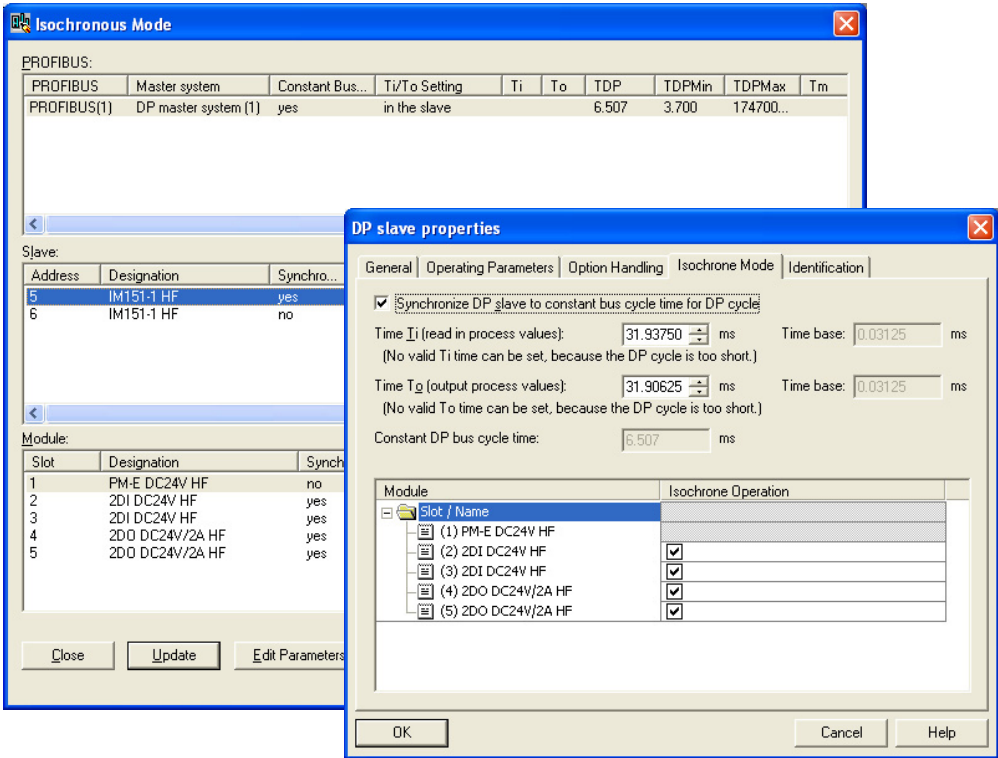


Fig. 20.15 Isochronous mode: Overview and DP slave properties

EDIT → MASTER SYSTEM → ISOCRONOUS MODE gives you an overview of all components involved in isochronous mode and the relevant parameters (Figure 20.15). If the checkbox “Times Ti and To same for all slaves” under “Network settings” and “Options” has been deactivated, you can set the update times individually for each slave (prerequisite for overlapping isochronous mode). Select the DP slave in the “Isochrone mode” window. “Edit Parameters” provides you with a dialog window for entering the individual update times and the modules involved in isochronous mode.

20.4.4 Addressing PROFINET IO

In a manner similar to how central modules are assigned to a CPU and controlled by it, the distributed modules with PROFINET IO (stations, IO devices) are assigned to an IO controller. The IO controller with all “its” IO devices is referred to as a PROFINET IO system.

Like central modules, the IO devices occupy addresses in the CPU’s I/O area (“logical address area”). The IO controller is so to say “transparent” for the addresses of the IO

devices; the CPU “sees” the addresses of the IO devices so that the addresses of the IO devices must not overlap with those of the central modules, not even with the addresses of further distributed modules.

Each node operated on the Industrial Ethernet has an IP address. In addition, an IO device is assigned a device name, a device number (node number), a geographical address (slot) and at least one diagnostics address (Figure 20.16).

MAC address

The MAC address is an address assigned to the device which is unambiguous. It consists of three bytes with the vendor identification and three bytes with the device identification. The MAC address is usually printed on the device, and is assigned to it during the configuration procedure (if not already done in the factory).

IP address

Each node on the Industrial Ethernet subnetwork which uses the TCP/IP protocol requires an IP address. The IP address must be unambig-

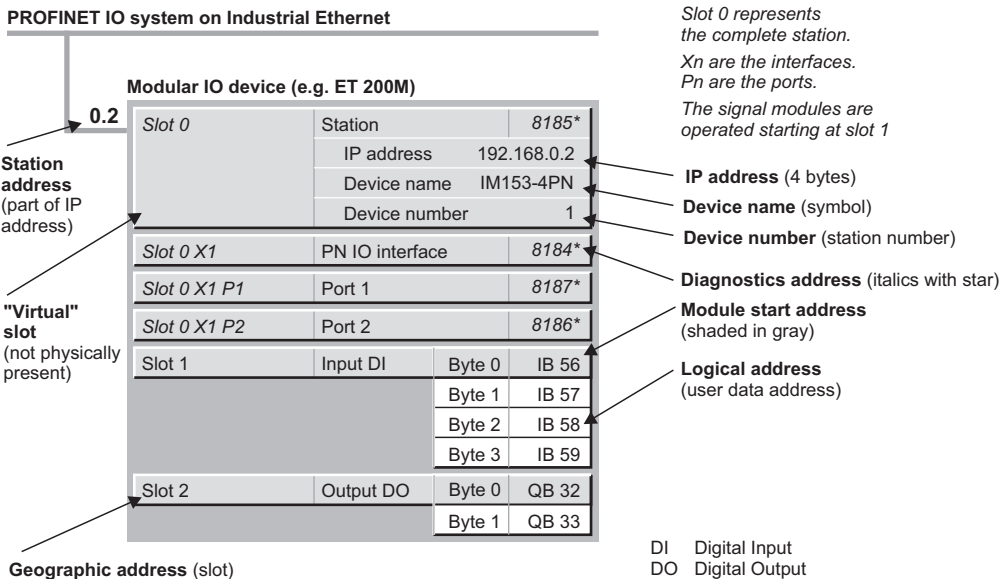
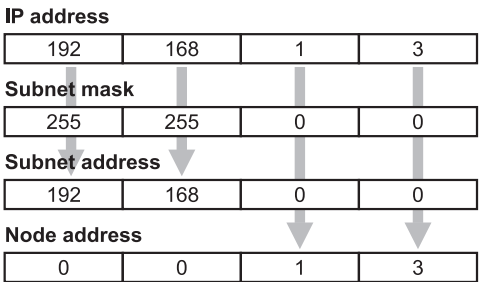


Figure 20.16 Addresses in a PROFINET IO system

uous on the subnetwork. It is assigned once to the IO controller for the nodes of a PROFINET IO system. Based on this, the hardware configuration assigns the IP addresses to the IO devices in ascending order.

The IP address is four bytes long, each being separated by a dot. Each byte is represented as a decimal number from 0 to 255.

The IP address comprises the address of the subnetwork and the address of the node. The subnetwork mask defines the share of the network address in the IP address. Like the IP address, it consists of four bytes which normally have a value of 255 or 0. Those bytes with a value of 255 in the subnetwork mask define the subnetwork address, the bytes with a value of 0 define the node address (Figure 20.17).



The subnet address is present left-justified in the IP address and is generated by ANDing the IP address with the subnet mask.

Figure 20.17 Example of IP address structure

Values other than 0 and 255 can also be assigned in a subnet mask, thereby dividing up the address area even further. For example, in a subnet 192.168 x x, the subnet mask 255.255.128.0 divides the stations into the two address areas 192.168.0.0 to 192.168.127.254 and 192.168.128.0 to 192.168.255.254.

Please note that the assignment of IP addresses is carried out in accordance with international, national, and company rules. For example, the IP addresses 10.0.0.0 to 10.255.255.255, 172.16.0.0 to 172.31.255.255 and 192.168.0.0 to 192.168.255.255 are provided for private net-

works in accordance with RFC 1918. These addresses are not passed on in the Internet.

The value 255 in the station address is envisaged for broadcasting. By using an address x.y.255.255, for example, all nodes in subnet x.y are addressed.

Device name, device number

During the configuration process, you assign the IO controller and each IO device a device name which must not be longer than 127 characters and consists of letters, digits, hyphens, and dots.

The name of the IO system can be appended to the device name, separated by a dot. To do so, check the "Use name in IO device/controller" checkbox in the properties of the PROFINET IO system.

Supplementary to the device name, the Hardware Configuration also assigns a device number to each IO device which is independent of the IP address and which you can change. You can use this device number (station number) to address the IO device from the user program, e.g. as an actual parameter on a system block.

Geographical address

The geographical address identifies a module slot. With central modules, the geographical address contains the number of the rack and that of the slot. With PROFINET IO, the geographical address contains the number of the PROFINET IO system, the station number, the slot number and possibly a subslot number.

The "virtual" slot 0 (not physically present) represents the IO device. From slot 1 onwards are the user data and diagnostics data. The system functions SFC 70 GEO_LOG and SFC 71 LOG_GEO are available for conversion from the geographical address to the logical addresses and vice versa.

Logical address, module starting address

You use the logical address to access the user data of a station. Each byte of the user data is unambiguously identified by a logical address. The logical address corresponds to the absolute address; a symbol (name) can be assigned to it so that it is easier to read (symbolic addressing).

The smallest logical address of a module or station is the module starting address (see also Chapter 1.4, “Module Addresses”).

Diagnostics address

Modules and stations which can deliver diagnostics data and do not have a user data address themselves are identified by the diagnostics address. The diagnostics address occupies one byte of peripheral input in the logical address volume. In the default setting, STEP 7 assigns the diagnostics address starting with the highest address in the I/O area of the CPU. You can change the diagnostics address. The address overview in the Hardware Configuration tool identifies the diagnostics address by a star.

In the example shown (Figure 20.18), the MPI/DP interface of the controller CPU 414-3 PN/DP is assigned the diagnostic address 8191 (not shown) and the PN interface is assigned address 8190. The diagnostic addresses for the ports and the PN IO system are assigned in subsequent order. The diagnostic addresses are assigned to the device CPU in a similar manner. From the viewpoint of the IO controller, the station of the device CPU is assigned the user data address of the first transfer area as the diagnostic address. It can then still address the station even if the parameters of the PN interface have been assigned by the device CPU (from the viewpoint of the controller CPU, no more diagnostic addresses are visible for the PN interface and the ports of the IO device).

Each compact and modular IO device is assigned the diagnostic addresses for the station, the PN interface, and the ports by the IO controller.

In the user program, the diagnostics data are scanned using system blocks. With a diagnostic interrupt, for example, the SFB 54 RALRM is used which reads the supplementary interrupt information. You can scan the diagnostics data record DS1 using the SFB 52 RDREC.

Transfer memory on an intelligent IO device

In the case of compact and modular IO devices, the addresses of the inputs and outputs are located together with the addresses for the central modules in the address volume of the controller CPU. Intelligent IO devices have a transfer

memory which can be divided into several transfer areas of different lengths. The individual areas then respond like modules whose lowest address is the module start address. From the viewpoint of the controller CPU, the intelligent IO device then appears as a compact or modular IO device, depending on the division.

When configuring the device, you can configure the individual areas of the transfer memory as inputs or outputs with the “module start address” and the area length. The addresses of the transfer memory must not overlap with the central modules in the device CPU. If the addresses are present in the process image, the areas can be handled by the user program like inputs and outputs, otherwise like peripheral inputs and outputs. If the device CPU possesses process image partitions, you can also assign a process image partition to each area.

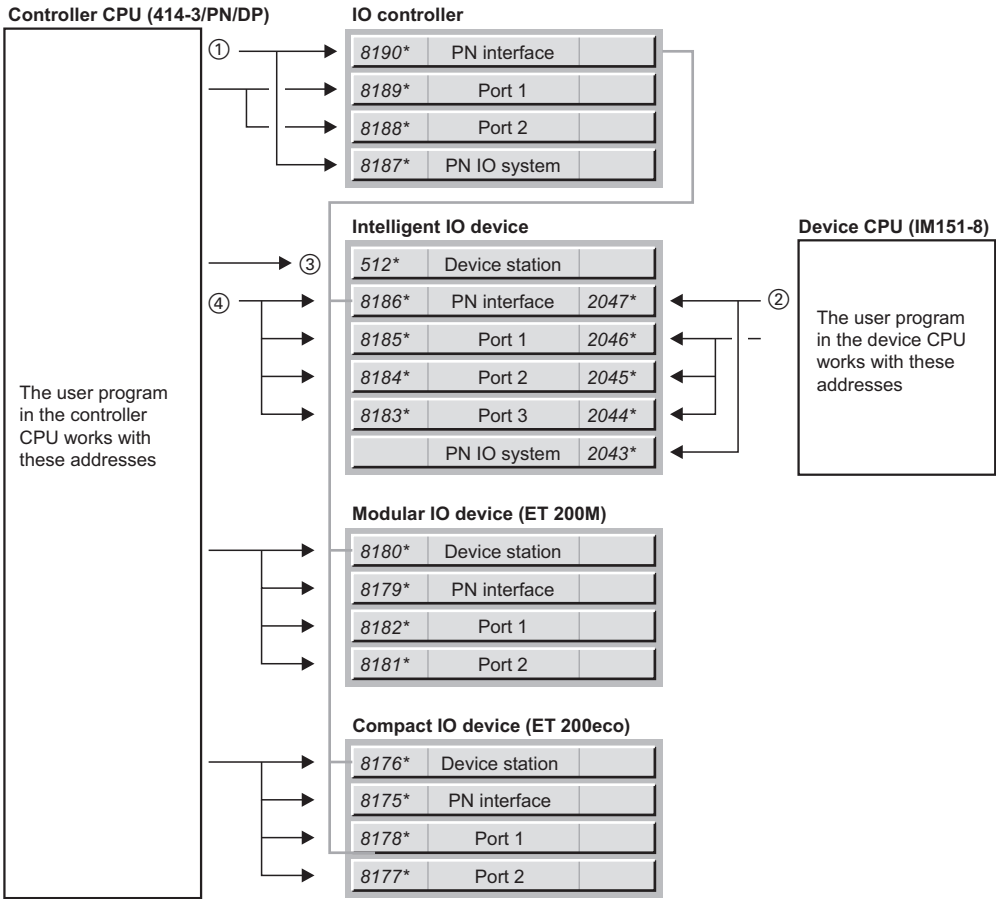
When coupling to the PROFINET IO system, you supplement the configuration at the controller end by the “module start addresses” from the viewpoint of the controller CPU and by the transmission direction. You assign inputs on the device side to outputs on the controller side and vice versa. If the addresses are present in the process image, the areas can be handled by the user program like inputs and outputs, otherwise like peripheral inputs and outputs. If the controller CPU possesses process image partitions, you can also assign a process image partition to each area. From the viewpoint of the controller CPU, the addresses of the transfer memory must not overlap with the addresses of other central or distributed modules in the controller station.

20.4.5 Configuring PROFINET IO

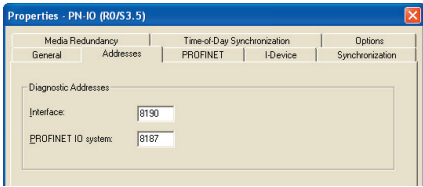
General procedure

You configure the distributed I/O as PROFINET IO essentially in the same way as the centralized modules. Instead of arranging modules in a mounting rack, you assign IO devices (nodes on the Industrial Ethernet subnetwork) to a PROFINET IO system. The following order is recommended for the necessary actions:

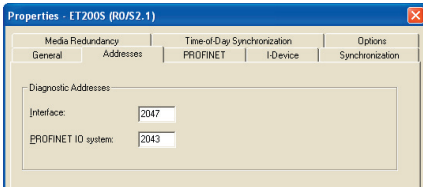
- 1) Create a new project or open an existing one with the SIMATIC Manager.



① Each bus interface has a diagnostics address which is set in the object properties of the interface in the "Addresses" tab (double-click on the "PN" line in the controller CPU).



② Each bus interface has a diagnostics address which is set in the object properties of the interface in the "Addresses" tab (double-click on the "PN" line in the device CPU).



③ The diagnostics address of the station of an intelligent IO device corresponds to the user data address ("module start address") of the first transfer area in the transfer memory.

④ The diagnostics address of the PN interface and of the ports of the intelligent IO device are only assigned by the IO controller if the checkbox "Parameter assignment for the PN interface and its ports on the higher-level IO controller" is activated in the device station.

Fig. 20.18 Diagnostics addresses in a PROFINET IO system

- 2) Create a Industrial Ethernet subnetwork in the project with the SIMATIC Manager.
- 3) Use the SIMATIC Manager to create the station in the project that is to accommodate the IO controller e.g. an S7-300 station.
- 4) If the project contains intelligent IO devices, you can now also create the corresponding device stations, for example an ET 200pro station.
- 5) You start the hardware configuration by opening the controller station. Insert a CPU with integrated PN interface. You assign the previously created Ethernet subnetwork to the PN interface and create a PROFINET IO system. In the "General" tab you can change the specified device name and IP address. You can also configure the remaining modules later. Save and compile the station.
- 6) If you have created an S7 station for an intelligent IO device, open this in the hardware configuration and insert an appropriate CPU with integrated PN interface, if applicable (with an ET 200 station, the CPU is inserted by STEP 7). Activate the "I device mode" of the PN interface, and configure the transfer memory from the viewpoint of the IO device. You can also configure the remaining modules later. Save and compile the station.

Subsequently generate a GSD file from the intelligent IO device and install it.

Proceed in the same manner with further stations envisaged for intelligent IO devices.

- 7) Now couple the IO devices to the PROFINET IO system. Open the controller station and drag the PROFINET node using the mouse from the hardware catalog to the PROFINET IO system, assign a device name, and possibly the device number (station number).
- 8) If you use IRT communication, create a new sync domain, accept the participating PROFINET IO systems, and set the properties of the relevant devices.
- 9) If you use IRT with the option "High performance", you must configure the network topology, either in the hardware configura-

tion directly at the port (interface connection) or centrally with the topology editor.

- 10) Save and compile all stations. The PROFINET IO system is now configured. You can now supplement the configuration with centralized modules, with IO systems, or with further IO devices. If you change the assignment of an intelligent IO device, you must also generate a new GSD file, and in turn a station from this, which you can add to the PROFINET IO system.

You can also represent the PROFINET IO system configured in this way graphically with the Network Configuration tool. Open Network Configuration by, for example, double-clicking on a subnetwork. Select VIEW → WITH DP SLAVES/IO DEVICES to display the IO devices. You can also create a PROFINET IO system (or more precisely, assign the nodes to an Ethernet subnetwork) with the Network Configuration tool. You parameterize the stations after opening them with the Hardware Configuration.

Before loading the configuration data, the device name must be assigned to each IO device ("naming"). In STOP mode, load the configuration data onto the CPU that accepts parameters such as the IP address. With the Hardware Configuration, load the data of the currently opened station (PLC → DOWNLOAD); with the Network Configuration, you can send the data to several stations, for example, with PLC → DOWNLOAD TO CURRENT PROJECT → STATIONS ON THE SUBNET.

At startup, the CPU transfers the configuring information to the IO devices and monitors the parameterization. The IO devices also receive their IP address with the parameters. Following successful parameterization, the user data are then exchanged cyclically between the IO controller and IO devices in RUN.

Configuring the IO controller

You must have created a project and an S7 station with the SIMATIC Manager. Using the hardware configuration you open the station and position a CPU module with a PN interface (see Chapter 2.3.1, "Arranging Modules")

To set the properties, double-click in the configuration table on the line with the PN inter-

face. You can change the device name of the IO controller on the "General" tab of the Properties window.

In order to assign an Ethernet subnet to the PN interface, click on the "Properties" button. Select the subnet in the displayed window, or create a new one using the "New" button. Adjust the IP address and possibly the subnet mask if necessary. Close the dialog box with "OK".

To create the PROFINET IO system, select the PN interface in the configuration table and then the command INSERT → PROFINET IO SYSTEM.

Mark the black/white rail, and select EDIT → OBJECT PROPERTIES. . Assign a name and an IO system number (from 100 to 115) on the "General" tab of the Properties window. You can also select here whether the system name is to be part of the device name for the IO controller and IO devices. You can access the S7 subnet ID using the "Properties..." button.

Configuring a compact IO device

The IO devices can be found in the hardware catalog under "PROFINET IO" in the corresponding subcatalog, e.g. "I/O". Select the desired IO device and drag it with the mouse to the PROFINET IO system.

Double-clicking on the IO device opens the Properties window. You can change the device name and number on the "General" tab. To change the IP address, click on the "Ethernet..." button.

The bottom part of the window shows the configuration table of either the selected IO device or the PROFINET IO system. You can switch over using the "Arrow" buttons.

In order to change a user data address, select the IO device and double-click on the address line in the configuration table. In the Properties window, you can then change the address on the "Addresses" tab.

Configuring a modular IO device

The IO devices can be found in the hardware catalog under "PROFINET IO" and the corresponding subcatalog, e.g. I/O.

Click on the required interface (basic module) and drag it using the mouse to the symbol for the PROFINET IO system. Double click on the IO device to obtain the properties sheet of the station on which you can set the device name and number. By double-clicking on the IO device, you are provided with the properties sheet of the station; you can set the device name and number on the "General" tab. You can change the proposed IP address after clicking on the "Ethernet..." button.

The bottom part of the window shows the configuration table of either the selected IO device or the PROFINET IO system. You can switch over using the "Arrow" buttons.

Now position the modules which you find in the hardware catalog *under the required interface (!)* in the configuration table. Double-clicking on the line opens the Properties window and allows the assignment of parameters to the module.

Special functions for ET 200S and ET 200pro

The *Combine modules* function optimizes the address assignments for modules with two or four bit channels. You can find a description of this function under , "Combining modules" on page 293.

With the *Option handling* function, you prepare a future extension of an ET 200S or ET 200pro station. You can find a description of this function under , "General handling of options" starting on page 294.

Configuring an intelligent IO device

You configure a station for an intelligent IO device like a controller station: In the SIMATIC Manager or in the hardware configuration, insert a new station into the project, position the CPU module with the PN interface, and connect the PN interface to the Ethernet subnet. Examples of intelligent IO devices you can use are a CPU 400 with firmware release V6.0 or higher, a CPU 300, a CPU ET 200S, or a CPU ET 200pro with firmware release V3.2 or higher.

To set the operating mode, select the PN interface in the configuration table and then EDIT → OBJECT PROPERTIES. Select the "I device" tab in the Properties window, and check the "I device mode" checkbox there.

If the "Parameter assignment for the PN interface and its ports on the higher-level IO controller" checkbox is selected, the IO controller assigns the parameters for the PN interface. Otherwise the parameters are set by the IO device.

You must subsequently still configure the transfer memory. The transfer memory is the user data interface between the IO controller and the IO device. It can be divided into several transfer areas whose addresses correspond to individual modules. With correspondingly designed CPU modules, you can also specify input modules of the IO device in the transfer area, which the IO controller can then access almost directly.

Following activation of the I device mode, you create a new transfer area in the transfer memory using the "New..." button. You define the type of transfer area in the transfer area's properties: "Application" if it is to be a freely-defined area and "I/O" if an I/O module is to be addressed.

When selecting "Application", you define whether the area is to be an input or output from the viewpoint of the IO device, and define the start address, the length (in bytes), and also the process image partition if required (Figure 20.19).

When selecting "I/O", you click on the "Select I/O" button and then select the desired input

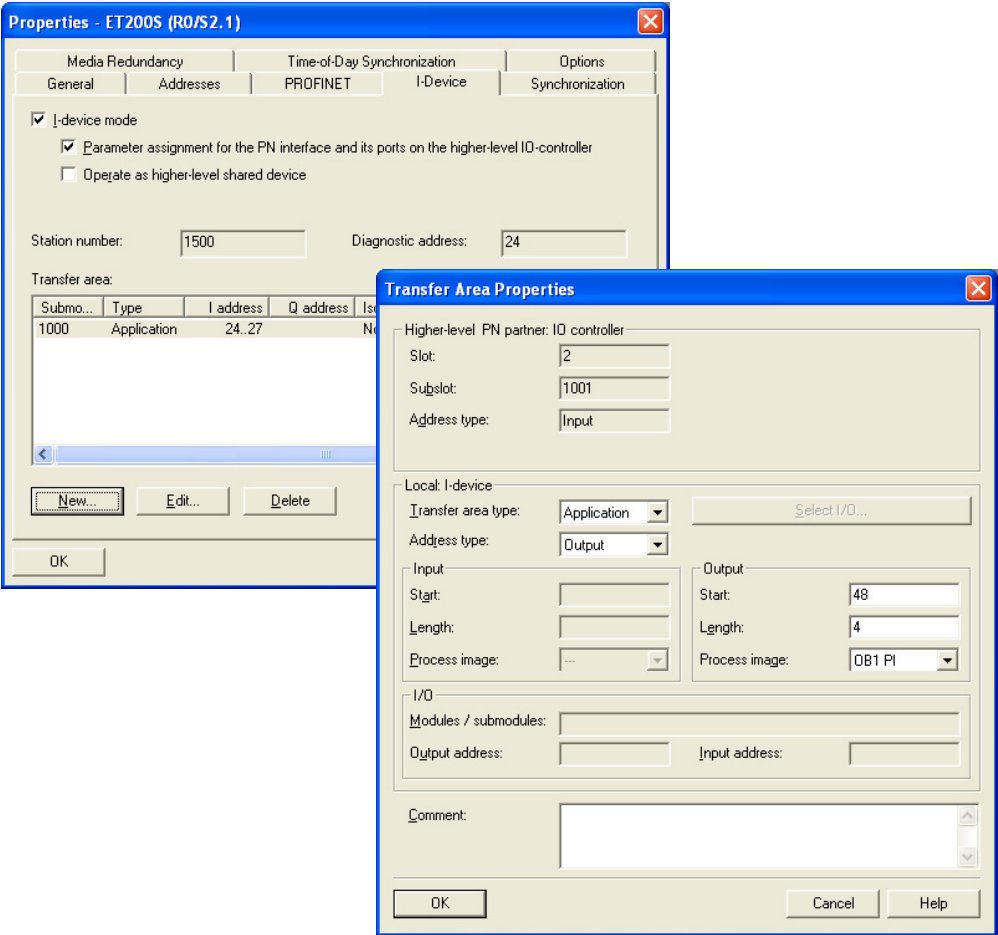


Fig. 20.19 Configuring the transfer memory of an intelligent IO device

module from the previously configured input modules in the dialog window. The I/O module appears automatically as the output transfer area whose start address (corresponds to the module start address) you can change.

Once you have configured all transfer areas, save and compile the station, and generate a GSD file with `OPTIONS → CREATE GSD FILE FOR I DEVICE ...` from the IO device. In the "Designation for I device proxy" box in the dialog window, you can change the name of the PN interface as it is to be displayed later in the IO controller.

Click the "Create" button. Following creation of the GSD file, you can save it using "Export" and install it later. To install it immediately, click on the "Install" button and then on "Close". STEP 7 creates a folder "Preconfigured Stations" in the hardware catalog under "PROFINET IO", and inserts a folder with the symbol for the intelligent IO device just created.

The intelligent IO device is now coupled as with a compact IO device: Open the controller station and drag the symbol for the intelligent IO device to the PROFINET IO system using the mouse.

With the IO device selected, the transfer areas are displayed in the configuration table as a subplot for slot 2 with the user data address as "seen" by the controller CPU. To change an address, double-click on the address line and enter the new address on the "Addresses" tab in the Properties window.

Configuring PN/PN coupler

The PN/PN coupler connects each of two Industrial Ethernet subnets to a PROFINET IO system. In each of the two PROFINET IO systems, one half of the PN/PN coupler appears as an IO device. Using Configuration of transfer memory, combine the two IO devices.

With a correspondingly designed PN/PN coupler, you can use other functions in addition to the cyclic I/O transmission, for example the shared device function (see section „Shared device“ auf Seite 314).

Prerequisite: The two PROFINET IO systems are set up. You have opened one of the controller stations.

You can find the symbol of the PN/PN coupler in the hardware catalog under "PROFINET IO" and "Gateway" in the "PN/PN coupler" folder. Use the mouse to drag one half of the PN/PN coupler (e.g. X1) to the PROFINET IO system. Drag the second half to the second PROFINET IO system.

To combine the two halves, select the PN/PN coupler and then `EDIT → OBJECT PROPERTIES`. Configure the coupling partner on the "Coupling" tab in the Properties window. Specify the other half of the PN/PN coupler here (Figure 20.20).

You can set the device name and number for each half of the coupler on the "General" tab in the Properties window – just like with an IO device – and change the IP address after clicking on the "Ethernet..." button.

With the PN/PN coupler selected, the configuration table is displayed in the bottom part of the window. Double-clicking on the station line (slot 0) or on the interface line opens the Properties window in which you can set the parameters of the station or of the PN interface for the current part of the PN/PN coupler.

Both halves of the coupler have a transfer memory for data exchange which can be divided into several transfer areas. Each area corresponds to a signal module whose addresses are within the address volume of the respective IO controller. Each area has a start address ("Module start address") and a length in bytes.

If an input area is configured in one half of the PN/PN coupler, an output area of the same length must be created in the other half, and vice versa. If the other subnet is in the same project, STEP 7 creates the associated transfer area in the other half of the coupler.

You configure the transfer areas like signal modules: Input and output areas of various lengths are listed in the hardware catalog under the corresponding PN/PN coupler. Use the mouse to drag the symbol with the desired property to a slot in the configuration table, starting without gaps at slot 1. A double-click on the line of the transfer area opens the Prop-

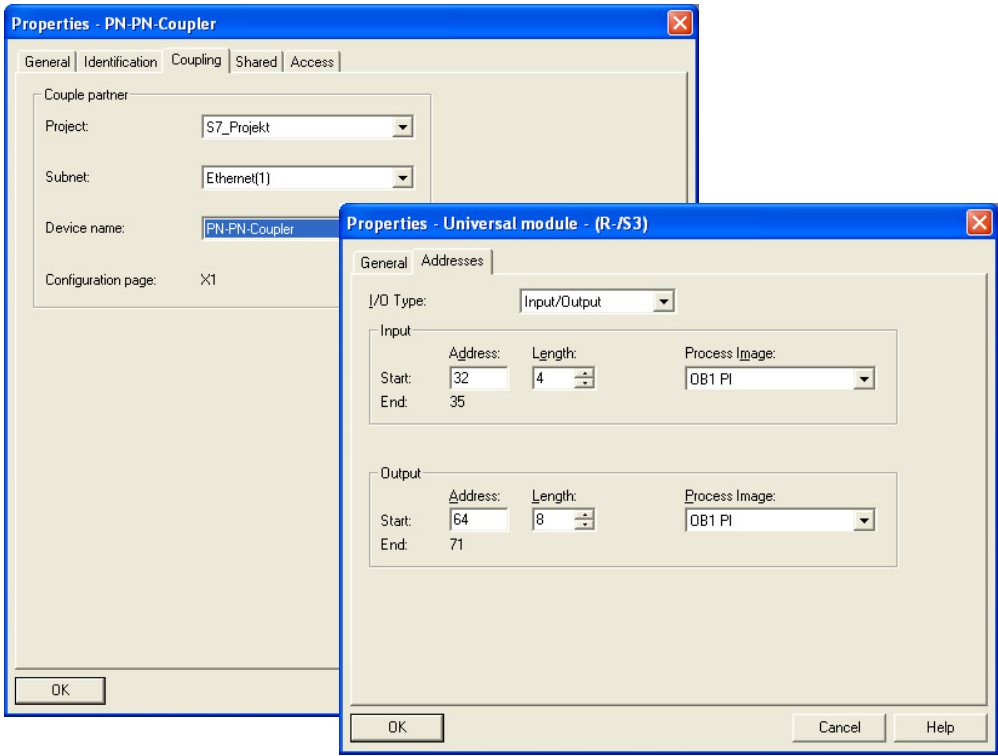


Fig. 20.20 Configuring a PN/PN coupler V3.0: Connection and characteristics of the transfer interface

erties window in which you can set the start address and the process image partition used on the "Addresses" tab.

If you are using the universal module, you must set the I/O type, the start address, the area length, and the process image partition in the properties (Figure 20.32).

On the "Connection" tab, you can also clear down the connection between the subnets again (enter " - - - -" under "Subnet").

Configuring the IE/PB link

You can find the IE/PB-Link PNIO in the hardware catalog under "PROFINET IO" and "Gateway" in the "IE/PB Link PN IO" folder. Use the mouse to drag the symbol for the IE/PB link to the PROFINET IO system. In the Properties window which is displayed, you set the PROFIBUS subnet and the station numbers in this subnet.

Double clicking on the symbol with the PROFINET IO system previously selected outputs the properties window in which you can set the connection to a PROFIBUS subnetwork and the node number in this subnetwork in the "Parameters" tab.

EDIT → OBJECT PROPERTIES with the IE/PB link selected displays its properties window with the facility for setting the device name and number on the Ethernet subnetwork. You can change the IP address using the "Ethernet..." button.

The IE/PB link is quasi the DP master for the "subordinate" DP master system. You position the DP slaves from the hardware catalog on this master system, and assign them with the desired properties (see Chapter 20.4.2, "Configuring PROFIBUS DP").

The DP slaves require a device number in order to address them over PROFINET IO. You can make the assignments between node number on

the PROFIBUS and device number on the PROFINET in the properties window of the IE/PB link in the "Device numbers" tab. As standard, STEP 7 uses the PROFIBUS addresses as the device number (indicated by a star on the device number). Select a DP slave in the list, and click the "Change" button.

The IE/PB link is able to pass on time-of-day telegrams and parameterization data records. You can make the settings for this in the properties window in the "Options" tab.

Configuring the IE/AS-i link

You can find the IE/AS-i Link PNIO in the hardware catalog under "PROFINET IO" and "Gateway" in the "IE/AS-i Link PN IO" folder. Use the mouse to drag the symbol for the IE/AS-i link to the PROFINET IO system.

EDIT→ OBJECT PROPERTIES with the IE/AS-i link selected shows its properties window with the facility for setting the device name and the device number on the Ethernet subnet. You can change the IP address with the "Ethernett...." button.

The IE/AS-i link effectively functions as the AS-i-Master. The AS-i master system with the AS-i slaves is not displayed as a subnet by the Hardware Configuration. If the IE/AS-i link is selected, you receive the configuration table in which you "plug in" the AS-i slaves that appear in the catalog under the link symbol.

The configuration of the AS-i master system is described in Chapter 20.4.2, "Configuring PROFIBUS DP" under "Configuring the DP/AS-i link".

20.4.6 Special Functions for PROFINET IO

GSD files

You can subsequently install IO devices which are not included in the module catalog. To do this, you require the type file tailored to the IO device (GSD file, Generic Station Description). Suitable for IO devices are GSD files with GSD version 5 or higher in XML format (GSDML, Generic Station Description Markup Language).

To install, select OPTIONS → INSTALL GSD FILES in the hardware configuration and enter

the directory of the GSD file or of a different STEP 7 project in the displayed window. STEP 7 accepts the GSD file and displays the IO device in the hardware catalog under "PROFINET IO" and "Additional Field Devices".

STEP 7 saves the GSD files in the directory ...\\Step7\\S7DATA\\GSD. The GSD files deleted when installing or importing at a later time are stored in the subdirectory ...\\GSD\\BKPx. From here, they can be restored with OPTIONS → INSTALL GSD FILE.

Shared device

The "Shared device" function allows different IO controllers to access submodules (I/O modules and transfer areas) in one IO device. The associated IO device is used by the IO controllers together (shared device). Each submodule of the shared device is assigned to an IO controller.

The basic conditions for use of a shared device are:

- ▷ The IO controller and the IO device must be present in the same Ethernet subnet.
- ▷ When using isochronous real-time communication (IRT), a shared device can only be used with the IRT option "High performance".
- ▷ The shared device function can only be used with "even" send cycle times.
- ▷ A shared device cannot be operated in an isochronous manner with the constant PROFINET IO cycle.

The shared device function is available with a CPU 400 with firmware version 6.0 and higher and with a CPU 300 or CPU ET 200 with firmware version 3.2 and higher.

Prerequisite: A project has been created with two or more IO controllers and PROFINET IO systems on the same Ethernet subnet.

To create a (modular) shared device, open a controller station and drag the IO device from the hardware catalog to the PROFINET IO system with the mouse. Configure the modules by dragging from the hardware catalog to the slot in the configuration table. Position all modules for all IO controllers.

Following configuration, copy the IO device into the clipboard, for example using the "Copy" command from the shortcut menu. Save the controller station, and open another one.

To insert the saved IO device, click with the right mouse button on the PROFINET IO system and select the "Shared insert" command from the shortcut menu. Subsequently save the controller station. IO devices of identical design are now present in the two controller stations. Repeat inserting with the other IO controllers if applicable.

Double-click on the IO device in one of the controller stations. The shared device (of the other PN IO system) is entered in the Properties window in the "Coupled devices" table on the "Shared" tab. Here you can also delete the coupling again: Select the shared device in the "Coupled devices" table and click on the "Uncouple" button. This tab also permits coupling of the same type of IO devices which were not transferred with the "Shared insert" command: Select the IO device in the "Devices which can be coupled" table and click on the "Couple" button.

To assign the modules to an IO controller, open the "Access" tab. All modules are listed in a tree structure. A module has the value "Full" if it is assigned to the IO controller of the currently open PROFINET IO system. Otherwise it has the value "---". Open the shared devices in succession in each PROFINET IO system, and assign the modules to the associated IO controller by clicking in the "Value" column (Figure 20.21).

If the IO controller is in a different project, you must manually configure the shared device in the other project with exactly the same module assignment, but with the assignment referring to the current IO controller. Save the controller station following the assignment.

Real-time communication with PROFINET

PROFINET IO offers several types of data transfer:

- ▷ Non-time-critical data such as configuration and diagnostics information is transferred

acyclically with the TCP/IP communication standard.

- ▷ User data (input/output information) is exchanged cyclically between the IO controller and the IO device (real-time RT) within a defined time period – the update time.
- ▷ Time-critical user data, for motion control applications, for example, is transferred with hardware support isochronously (isochronous real time IRT).

A permanent communication channel is reserved on the Ethernet subnet for IRT communication. RT communication – cyclic data exchange between the IO controller and IO devices – and non-real-time TCP/IP communication take place in parallel within the update time. In this way, all three communication types can exist in parallel on the same subnet.

Send cycle time/update time

Cyclic data exchange is handled within a specific time frame, the send cycle time. STEP 7 calculates the send cycle time from the configuration information on the PROFINET IO system. The send cycle time is the shortest possible update time. This is the time period within which each IO device in the IO system has exchanged its user data with the IO controller. The actual update time for an IO device can be a multiple of the send cycle time. You can increase the update time manually, in order to reduce the bus load, for example.

The update time is the same length for all IO devices in the IO system as standard. Under certain circumstances, you can reduce the update time for individual IO devices if you increase the time for other devices whose user data can be exchanged non-time-critically.

You can configure the send cycle time (without IRT communication) centrally in the Properties dialog box of the PN interface on the "PROFINET" tab, or in the Properties dialog box of the PROFINET IO system on the "Update time" tab.

This tab also lists the IO devices with the update times. You can increase the time for an IO device by selecting it and clicking on the "Edit" button, or you can set the time in the Properties

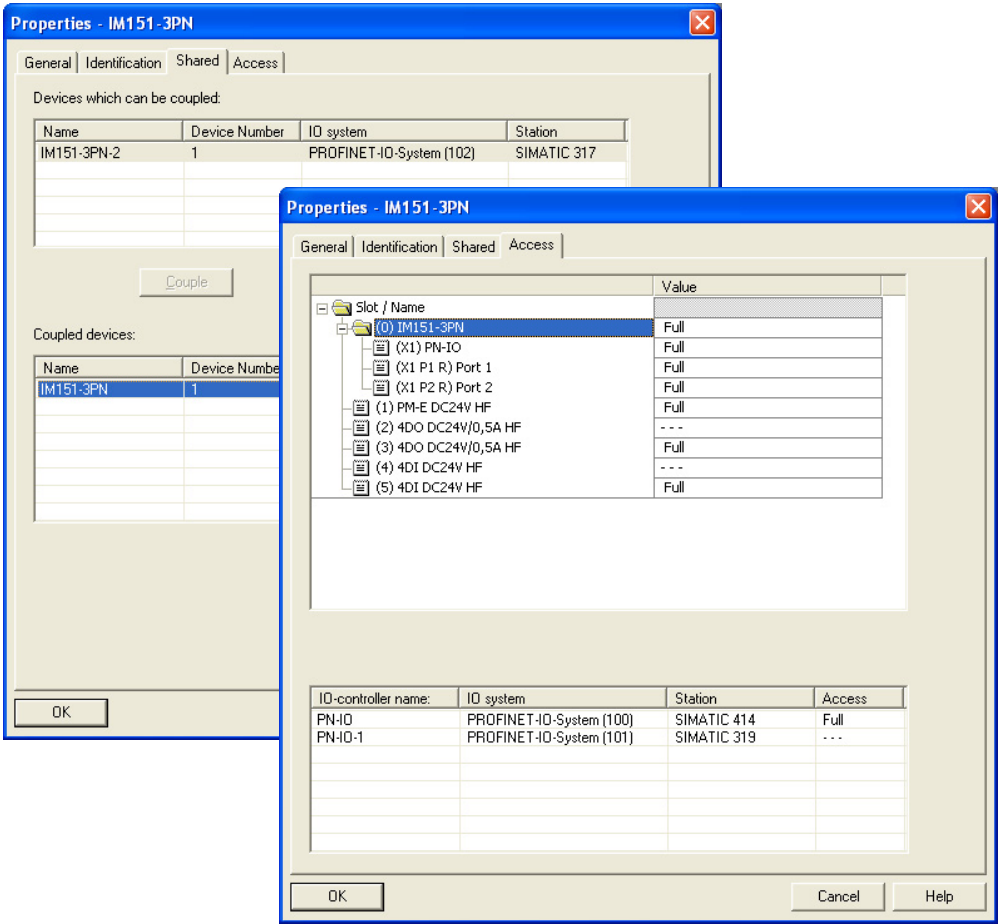


Fig. 20.21 Configuring a shared device

dialog box of the PN interface of the IO device on the "IO cycle" tab.

In addition to the update time, you can set the watchdog timer in the interfaces properties. This is the product of the update time and the "Number of accepted update cycles with missing IO data".

If there is at least one synchronized device in the IO system, the send cycle time is determined by the sync master of the sync domain and can only be modified in the properties of the sync domain. Select the IO system in the Hardware Configuration or the subnet in the Network Configuration. Select EDIT → PROFINET IO → DOMAIN MANAGEMENT. The cy-

cle time can then no longer be modified anywhere else.

Real time

Real time (RT) means that a system processes external events within a defined time. If it responds predictably, it is called deterministic. In RT communication, transfer takes place at a specific time (send cycle time) within a defined interval (update time). PROFINET IO allows the use of standard network components for RT communication.

If not all data to be exchanged is transferred within the planned time frame, due, for example, to the addition of new network compo-

nents, some data is distributed to other send frequencies. This can result in an increase in the update time for individual IO devices.

Isochronous real time

Isochronous real-time (IRT) is hardware-supported real-time communication designed, for example, for motion control applications. IRT message frames are deterministically transmitted via planned communication paths in a specified order. IRT communication therefore requires network components that support this planned data transmission.

Isochronous real-time is available in the "High flexibility" option for simple configuration and plant expansions and in the "High performance" option for fast updating times.

To be able to configure IRT communication, set up a new sync domain (see next section) and determine a sync master to handle the synchronized distribution of the IRT message frames to the sync slaves. IRT with the "High performance" option requires a topology configuration (see section "Topology Editor") and thus a defined structure that takes account of the transmission properties of the cables and the switches used.

Sync domain

A sync domain is a group of PROFINET IO nodes that exchange synchronized data with each other. One node (this can be an IO controller or an IO device) assumes the role of the sync master, and the others are the sync slaves.

A sync domain can contain several IO systems, but an IO system is always assigned entirely to one single sync domain. Several sync domains can exist on one Ethernet subnet.

When an IO system is configured, a special sync domain is automatically created: the *sync-domain-default*. All configured IO systems, IO controllers and IO devices are first located in the sync domain *syncdomain-default*.

Create a new sync domain for IRT communication, and transfer the IO system (from the *sync-domain-default*) to the new sync domain. Not all devices of an IO system have to be synchronized, or in other words, exchange data with IRT communication. When configuring, the

unsynchronized nodes are first also managed in the sync domain; at runtime only the synchronized nodes remain in the sync domain.

Configuring a new sync domain

Prerequisite: You have configured the Ethernet subnet with one or more PROFINET IO systems. The nodes involved in IRT communication must also support this function.

To create a new sync domain, select the PROFINET IO system in the Hardware Configuration or the subnet in the Network Configuration, and select EDIT → PROFINET IO → DOMAIN MANAGEMENT. The "Domain Management" window that appears shows the sync domain *sync domain default* and all IO systems on the same subnet that are located in the sync domain (Figure 20.22).

Create a new sync domain with the "New" button, assign a name, and select an IO system with the "Add" button. If you want to add further IO systems, repeat the procedure. IO systems added to the new sync domain are no longer a component part of the sync domain *sync-domain-default*.

The IO systems of the sync domain and the bus nodes of the selected IO system are displayed. Now select a device, then "Device Properties", and set the synchronization type (sync master or sync slave) and the RT class (RT, IRT "High flexibility" or IRT "High performance") in the parameters in the Properties window. Proceed appropriately with the other bus nodes. Only one device can be the sync master, and all other devices are sync slaves. A sync domain may contain either devices of the classes IRT "High flexibility" and RT or the classes IRT "High performance" and RT.

Select the send cycle time in the "Send cycle time" field, and select the proportion of bus communication reserved for IRT with the "Details" button. Confirm with "OK" to save the settings.

Topology Editor

The Topology Editor allows cabling configuration of devices on the Industrial Ethernet subnet. The logical connections between the PROFINET devices are configured with the configur-

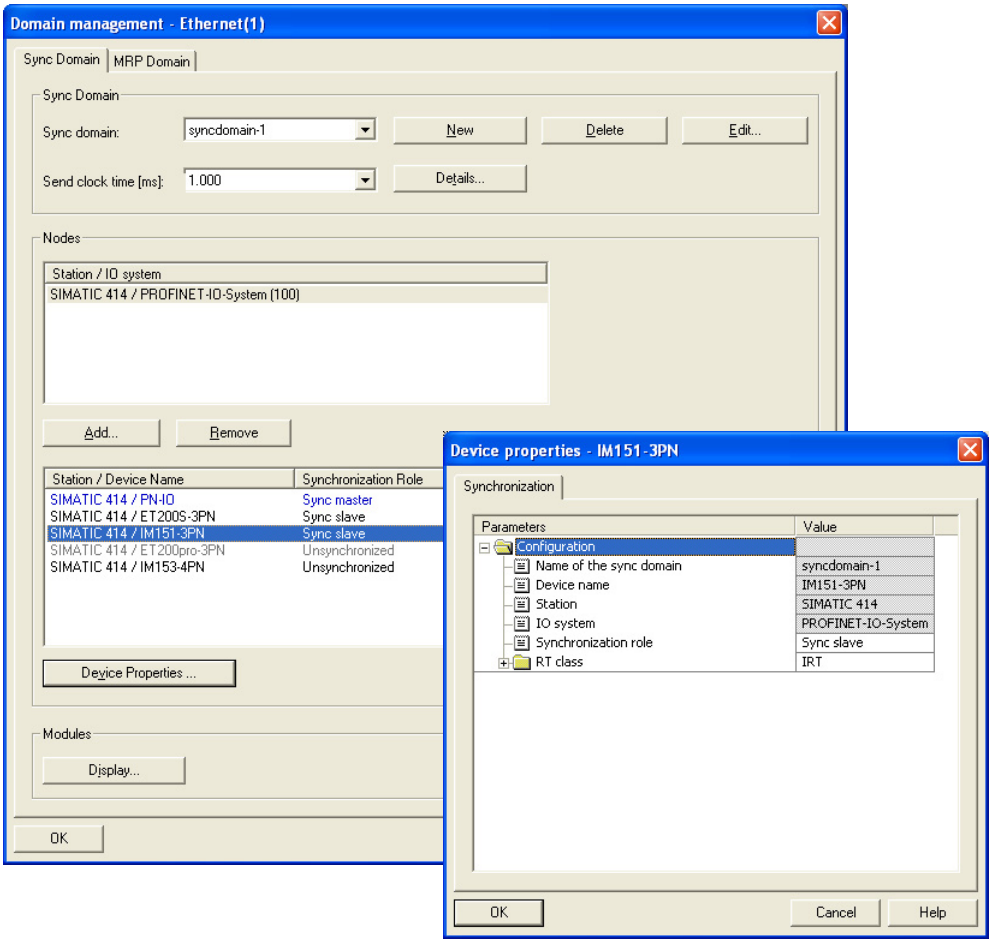


Fig. 20.22 Configuring a new SYNC domain

ing tools Hardware Configuration and Network Configuration. The Topology Editor is used to configure the physical connections with the properties Length and Cable type for determining the signal runtimes. Use of the Topology Editor is a prerequisite for using IRT communication (isochronous real-time) with the "High performance" option.

The physical connections between devices on the Ethernet subnet are point-to-point connections. The connections on a PN interface are called ports. The Ethernet cable connects a device port to a port on the partner device.

To enable several nodes to communicate with each other, they are connected to a switch that

has several connections (ports) and that distributes signals. There are also S7 devices featuring a PN interface that has two or more ports separated by an integral switch. With this interface you can cable communication devices in a linear bus topology without external switches.

You can configure the connection of two ports with the Hardware Configuration. In the configuration table, select the port and then EDIT → OBJECT PROPERTIES. On the "Topology" tab in the Properties window, you can now determine the partner port and edit the properties of the cable.

Before you call the Topology Editor, use the Hardware Configuration or the Network Confi-

guration to configure the communication partners on the Ethernet subnet including the necessary switches. In the Hardware Configuration, open the S7 station with the IO controller, select the PROFINET IO system and then select EDIT → PROFINET IO → TOPOLOGY. In the Network Configuration, select an Ethernet subnet and then EDIT → PROFINET IO → TOPOL-OGY.

The *table view* shows the port pairs of all configured active and passive components in the interconnections table. Using a filter, you can set the display of all ports, of only the connected ports, or of only the unconnected ports (Figure 20.23).

You can set the connection to the partner port in the object properties of the port. You can clear down an interconnection by selecting the port

and right-clicking on DISCONNECT PORT INTER-CONNECTION .

If there is a connection to the plant, you can use the “Online” button to check whether the devices configured offline are available and what their status is. The comparison is based on the device name, the IP address and the device ID. The data determined online is displayed in the "Status" and "Attenuation value" columns.

In the *graphic view*, the topology window displays the devices, their ports and the interconnection. The configured devices are displayed in the offline view, and the actual devices available in the plant are displayed in the online view, provided there is an online connection to the plant.

To facilitate editing, you can “close” the representation of the stations, hide the thumbnail and

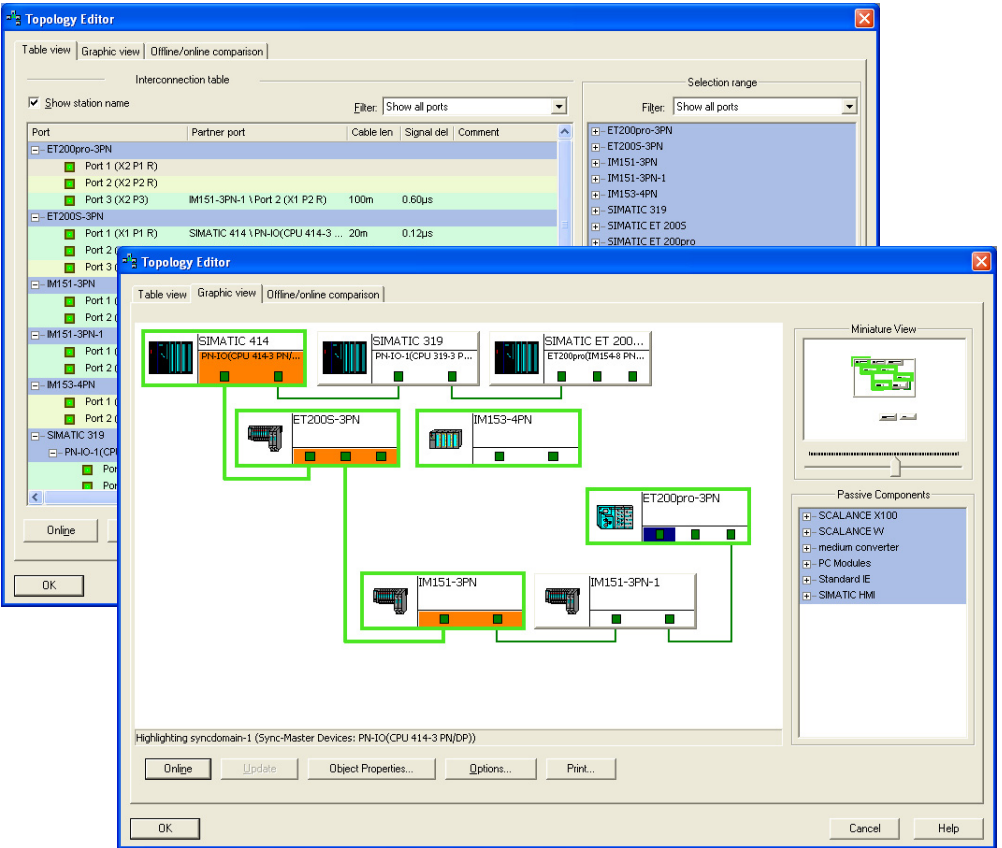


Figure 20.23 Tabular and graphical view of the Topology Editor

the catalog of the passive components ("Options" button and "Options" tab), and enlarge or reduce the view by scrolling with the middle mouse key.

To interconnect two ports, select a port and drag a connection to the partner port while holding down the right mouse key. You can disconnect a port by selecting the connection with the right mouse key and then selecting DISCONNECT PORT INTERCONNECTION.

On the *Offline/Online comparison* tab, the topology configured offline is shown alongside the topology determined online for comparison purposes. All the stations and modules are displayed with their ports, relevant partner ports, and cable data. In this way, you can check the configuration with the connections and cables, and supplement missing system components if required.

A selection can be made with the relevant filter settings. In the overviews the determined differences, e.g. the modules that the Topology Editor could not assign, are highlighted in color. You can now undertake manual assignment.

Isochronous mode

The "Isochronous mode" function permits synchronous input, processing, and output of I/O signals in a fixed (equidistant) time pattern. A prerequisite for isochronous mode is isochronous realtime (IRT) with the "High performance" option.

The basis of the time pattern is the cycle time and the data cycle derived from this (the update time, Figure 20.24).

The data cycle is the interval at which the IRT transmission takes place on the subnet. The application cycle is the interval at which the isochronous mode OB is called. This is a multiple of the data cycle.

Ti is the time required for reading the I/O signals. It includes the times for preparation of the I/O signals in the input modules or electronic modules, and for processing in the IO device.

Ti is followed by the data cycle. This begins with transmission of the I/O signals over the subnet. Transmission takes place in both directions; the input signals are transmitted to the

Operating mode 1: The execution time of the isochronous mode program is shorter than one data cycle.

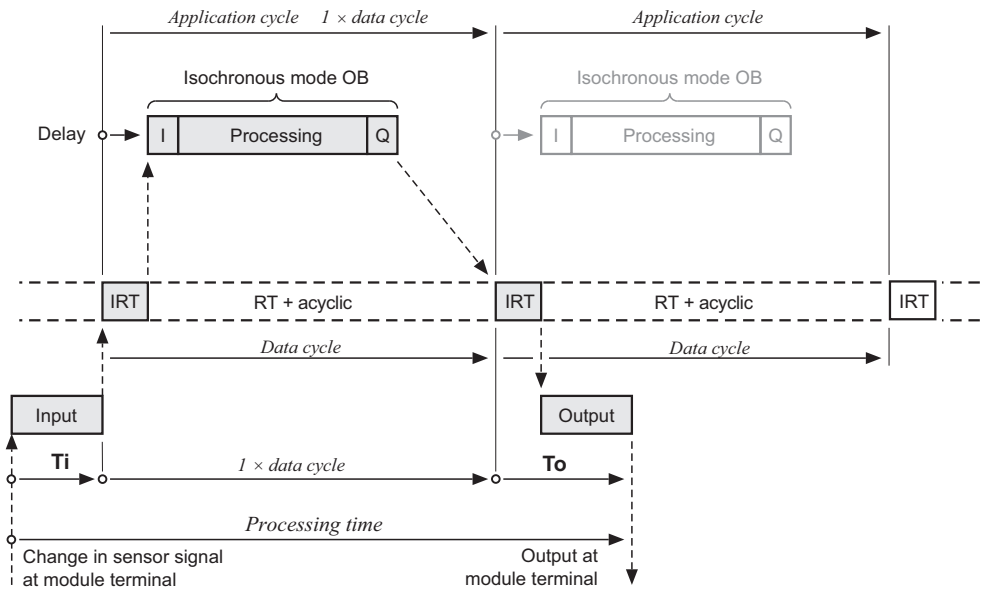


Fig. 20.24 Isochronous mode in the PROFINET IO system (application cycle = data cycle)

Operating mode 1: The execution time of the isochronous mode program is shorter than one data cycle.

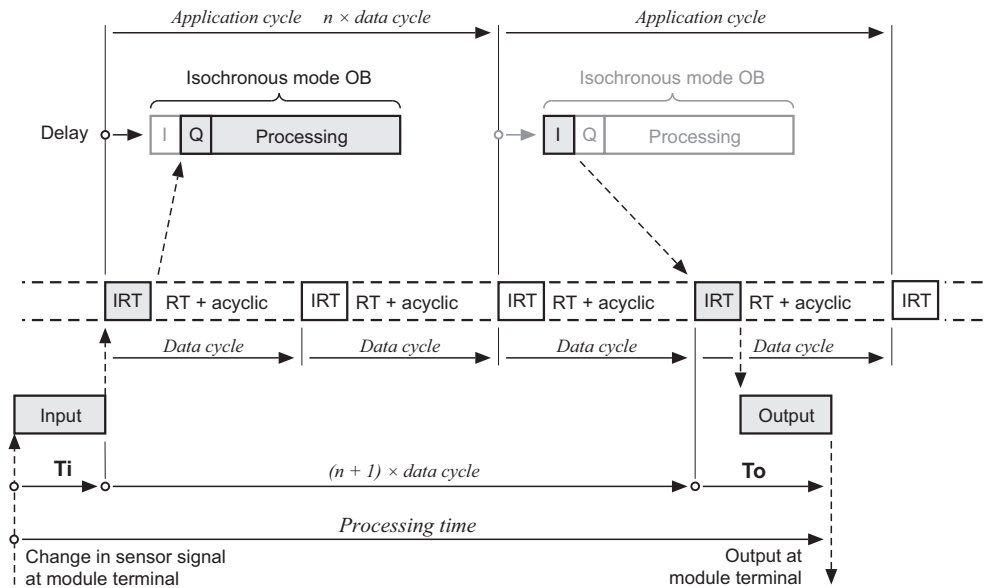


Fig. 20.25 Isochronous mode in the PROFINET IO system (application cycle > data cycle)

controller station, and the output signals (from the previous application cycle) are transmitted to the IO devices.

The isochronous mode organization block assigned to the PROFINET IO system is called following a delay time during which the IRT transmission takes place. System function SFC 126 SYNC_PI must be called in the organization block in order to read the input signals in isochronous mode, and system function SFC 127 SYNC_PO in order to write the output signals in isochronous mode. The processing time of the isochronous mode OB must be (significantly) shorter than the application cycle time, for the main program is further processed during the differential time.

To begins at the end of the data cycle. This is the time required for output of the I/O signals. It is made up of the transmission time on the subnet, the time for processing in the IO device, and the times for preparation of the I/O signals in the output modules or electronic modules.

With isochronous mode, a distinction is made between two types: The processing time of the isochronous mode program is (significantly)

shorter than the time for one data cycle, or it is longer. In the first case, the isochronous mode OB can be called in every data cycle (shown in Figure 20.36); in the second case, the cycle in which the isochronous mode OB is called – the application cycle – is a multiple of the data cycle (shown with factor 2 in Figure 20.25).

If the isochronous mode OB is called in every data cycle – the "Application cycle factor" is then 1 – the system function SFC 126 SYNC_PI for isochronous updating of the input signals is called first in the isochronous mode program. This is followed by processing of the signals, and subsequent output with the system function SFC 127 SYNC_PO.

With this mode, the shortest response time between an input signal and the corresponding output signal is therefore the total of T_i , the data cycle time, and T_o . The longest response time occurs if the input signal changes shortly after the time for reading-in, and is the total of T_i , T_o , and twice the data cycle time.

With an application cycle which takes longer than the data cycle (Figure 20.25), you should select a different sequence for updating of the

process image: Updating of the output signals first, then of the input signals, and then the processing. In this manner it is possible that the output signals are transmitted with the next possible data cycle (in the next application cycle) even if the data cycle time is short compared to the process image updating time.

With this mode, the shortest response time between an input signal and the corresponding output signal is therefore the total of T_i , the application cycle time, the data cycle time, and T_o . The longest response time occurs if the input signal changes shortly after the time for reading-in, and is the total of T_i , T_o , the data cycle time, and twice the application cycle time.

In order to configure isochronous mode, you create a PROFINET IO system with the con-

troller station and the IO devices, import the stations into a SYNC domain with the IRT option "High performance", and configure networking between the stations using the Topology Editor. In the device stations, you assign a process image partition, for example the TPA1, to the modules in their properties on the "Addresses" tab.

You assign the isochronous mode organization block to the PROFINET IO system in the CPU properties: Open the controller station and double-click on the CPU module to open the CPU properties window; select the "Synchronous cycle interrupts" tab and set the PROFINET IO system for the organizational block, for example the IO system number 100 for OB 61. Click on the "Details" button (Figure 20.26).

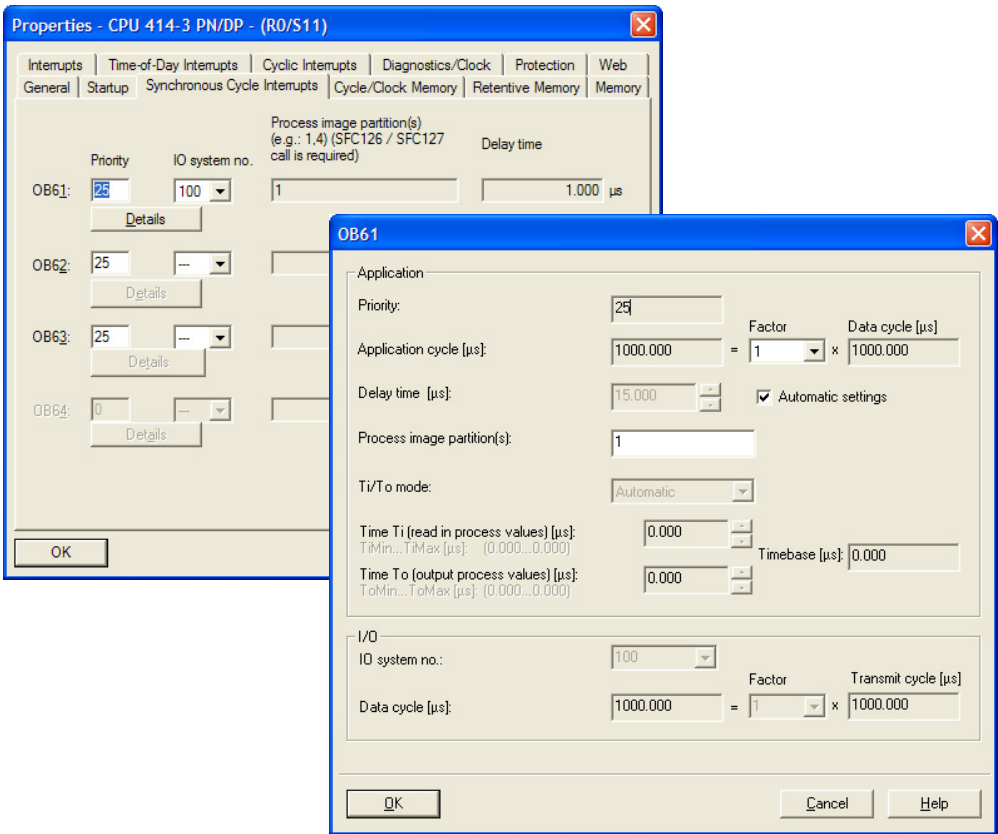


Fig. 20.26 Configuration of isochronous mode OB for PROFINET

The duration of the application cycle is calculated from the data cycle, multiplied by a factor which you specify on this tab. It is therefore necessary to estimate the processing time of the isochronous mode program and to compare this with the data cycle time.

If applicable, you set the delay time on this tab with which the isochronous mode OB is to start, and assign the process image partition which you have set for the module addresses in the IO devices. The following methods are available for determining the times T_i and T_o :

- ▷ "Automatic" – STEP 7 determines the times and sets them the same for all IO devices
- ▷ "Fixed" – you enter the times which then apply to all IO devices
- ▷ "In IO device" – the times are then set individually in the respective IO device

To assign the modules to isochronous mode, select the IO device, double-click on the PN interface in the configuration table, and select the "IO cycle" tab in the Properties dialog. In the section "Isochronous mode", assign the isochronous mode OB to the IO device, and click on the "Isochronous module/submodule..." button. You can activate or deactivate the individual modules of the IO device for isochronous mode in the displayed window. Proceed in the same manner for the other IO devices.

You are provided with an overview of the configuration if, with the controller station selected, you then select the EDIT → PROFINET IO → ISOCHRONOUS MODE command.

20.4.7 System blocks for distributed I/O

Read and write I/O signals

The following blocks transmit I/O signals to and from stations of the distributed I/O:

- ▷ FB 20 GETIO
Read all inputs of a station
- ▷ FB 21 SETIO
Write to all outputs of a station
- ▷ FB 22 GETIO_PA
Read some inputs of a station

- ▷ FB 23 SETIO_PA
Write to some outputs of a station
- ▷ SFC 14 DPRD_DAT
Read user data
- ▷ SFC 15 DPWR_DAT
Write user data

You can find the parameters of the blocks in the Tables 20.9 (FBs) and 20.10 (SFCs).

The loadable function blocks FB 20 to FB 23 have interfaces compliant with PI (PROFIBUS International) and can be used in conjunction with DP standard slaves and IO devices.

You can find the blocks in the *Standard Library* supplied with STEP 7 in the program *Communication Blocks*.

You can find the system functions SFC 14 and SFC 15 in the *Standard Library* supplied with STEP 7 in the program *System Function Blocks*.

FB 20 GETIO

Read all inputs of a station

FB 20 GETIO uses SFC 14 DPRD_DAT to consistently read all input data of a DP standard slave or an IO device or all data of an input area in the case of modular stations. The right-hand word of the ID parameter contains the start address of the input area to be read.

The destination area specified with the INPUTS parameter must be exactly the same length as the configured length of the input area read that is also output with the LEN parameter.

FB 21 SETIO

Write to all outputs of a station

FB 21 SETIO uses SFC 15 DPWR_DAT to consistently write all output data or all data of an output area of a DP standard slave or an IO device in the case of modular stations. The right-hand word of the ID parameter contains the start address of the output area to be written to.

The source area specified with the OUTPUTS parameter must be exactly the same length as the configured length of the output area to be written to. This is why information in the LEN parameter is irrelevant.

Table 20.9 Function block parameters for accessing the distributed I/O

FB	Parameter	Declaration	Data Type	Contents, Description
20	ID	INPUT	DWORD	Logical user data start address
	STATUS	OUTPUT	DWORD	Error information of SFC 14 DPRD_DAT ¹⁾
	LEN	OUTPUT	INT	Amount of bytes to be read
	INPUTS	IN_OUT	ANY	Destination area for the user data read (only data type BYTE permissible in the ANY pointer)
21	ID	INPUT	DWORD	Logical user data start address
	LEN	INPUT	INT	irrelevant
	STATUS	OUTPUT	DWORD	Error information of SFC 15 DPWR_DAT ¹⁾
	OUTPUTS	IN_OUT	ANY	Source area for the user data to be written (only data type BYTE permissible in the ANY pointer)
22	ID	INPUT	DWORD	Logical user data start address
	OFFSET	INPUT	INT	Number of the first byte to be read (from number 0)
	LEN	INPUT	INT	Amount of bytes to be read
	STATUS	OUTPUT	DWORD	Error information of SFC 81 UBLKMOV ¹⁾
	ERROR	OUTPUT	BOOL	Error occurred at signal state “1”
	INPUTS	IN_OUT	ANY	Destination area for the user data read (only data type BYTE permissible in the ANY pointer)
23	ID	INPUT	DWORD	Logical user data start address
	OFFSET	INPUT	INT	Number of the first byte to be written (from number 0)
	LEN	INPUT	INT	Amount of bytes to be written
	STATUS	OUTPUT	DWORD	Error information of SFC 81 UBLKMOV ¹⁾
	ERROR	OUTPUT	BOOL	Error occurred at signal state “1”
	OUTPUTS	IN_OUT	ANY	Source area for the user data to be written (only data type BYTE permissible in the ANY pointer)

¹⁾ Contains error information of the SFC used in the form DW#16#40xx xx00

FB 22 GETIO_PA

Read some inputs of a station

FB 22 GETIO_PA uses SFC 81 UBLKMOV to consistently read some of the input data or some of the data of an input area of a DP standard slave or an IO device in the case of modular stations. The right-hand word of the ID parameter contains the start address of the input area, the OFFSET parameter contains the number of the first byte to be read, and the LEN parameter contains the number of bytes.

The input bytes to be read must be addressed in the process image of the input in order to use the FB 22 GETIO_PA. If possible, it is preferable to use a partial process image. Please ensure that the OFFSET and LEN parameters do not violate any boundaries with neighboring data of other stations.

If the destination area specified by the INPUTS parameter is smaller than the input area read, the function only transfers as many bytes as can be written to the destination area. If the destination area is larger, only the first LEN bytes of the area are written to. In both cases, no error is indicated on the ERROR parameter. ERROR only has signal state “1” if an error is reported when calling SFC 81 BLKMOV.

FB 23 SETIO_PA

Write some outputs to a station

FB 23 SETIO_PA uses SFC 81 UBLKMOV to consistently write some of the output data to a DP standard slave or an IO device, or some of the data of an output area in the case of modular stations. The right-hand word of the ID parameter contains the start address of the output area,

Table 20.10 Parameters of the SFCs for consistent transfer of user data

SFC	Parameter	Declaration	Data type	Assignment, description
14	LADDR	INPUT	WORD	Configured start address (from the I area)
	RET_VAL	RETURN	INT	Error information
	RECORD	OUTPUT	ANY	Destination area for the read user data
15	LADDR	INPUT	WORD	Configured start address (from the Q area)
	RECORD	INPUT	ANY	Source area for the user data to be written
	RET_VAL	RETURN	INT	Error information

the OFFSET parameter contains the number of the first byte to be written, and the LEN parameter contains the number of bytes.

The output bytes to be written must be addressed in the process image of the output in order to use the FB 23 SETIO_PA. If possible, it is preferable to use a partial process image. Please ensure that the OFFSET and LEN parameters do not violate any boundaries with neighboring data of other stations.

If the source area specified by the OUTPUTS parameter is smaller than the output area to be written, the function only transfers as many bytes as the source area contains. If the source area is larger, only the first LEN bytes are transferred. In both cases, no error is indicated on the ERROR parameter. ERROR only has signal state “1” if an error is reported when calling SFC 81 BLKMOV.

SFC 14 DPRD_DAT

Read user data

SFC 14 DPRD_DAT reads consistent user data with a length of 3 bytes or greater than 4 bytes from a DP slave or an IO device. You specify the length of the data consistency when you parameterize the station (Table 20.10).

The LADDR parameter contains the module start address of the user data (input area). The RECORD parameter writes to the area in which the read data are stored. Variables of data type ARRAY and STRUCT or an ANY pointer of data type BYTE (e.g. P#DBzDBXy x BYTE nnn) are permissible as actual parameters.

Please note: If peripheral inputs (PI) are addressed whose addresses are in the process image (I), the process image is not updated.

SFC 15 DPWR_DAT

Write user data

SC 15 DPWR_DAT writes consistent user data with a length of 3 bytes or greater than 4 bytes to a DP slave or an IO device. You specify the length of the data consistency when you parameterize the station.

The LADDR parameter contains the module start address of the user data (output area).

The RECORD parameter writes to the area from which the transmitted data are read. Variables of data type ARRAY and STRUCT or an ANY pointer of data type BYTE (e.g. P#DBzDBXy x BYTE nnn) are permissible as actual parameters.

If peripheral outputs (PQ) are addressed whose addresses are in the process image output (Q), the process image is updated as with the transfer instruction (STL) or the MOVE box (LAD, FBD).

Activate/deactivate distributed station

The following system function activates or deactivates a station of the distributed I/O (DP slave or IO device):

▷ SFC 12 D_ACT_DP

Activate/deactivate distributed station

The parameters of this system function are shown in Table 20.11.

SFC 12 D_ACT_DP

Activate/deactivate distributed station

SFC 12 D_ACT_DP deactivates and activates stations of the distributed I/O and allows scanning of the deactivated or activated status. A distributed station can be a DP slave or an IO device.

Table 20.11 Parameters of the SFC for activation and deactivation of I/O stations

SFC	Parameter	Declaration	Data type	Assignment, description
12	REQ	INPUT	BOOL	Request for activation/deactivation if REQ = "1"
	MODE	INPUT	BYTE	Function mode 0 Scan whether the station is activated or deactivated 1 Activate station 2 Deactivate station
	LADDR	INPUT	WORD	Any logic address of the station
	RET_VAL	RETURN	INT	Result of scan or error information
	BUSY	OUTPUT	BOOL	Job still running if BUSY = "1"

SFC 12 D_ACT_DP is called in the cyclic program; a call in the start-up routine is not supported. The SFC works in asynchronous mode, i.e. processing of a job can extend over several program cycles. "1" in the REQ parameter starts an activation or deactivation job. As long as the BUSY parameter has the signal state "1", the REQ parameter must remain occupied by "1". The job is finished when BUSY = "0".

After deactivation, a configured (and existing) station is no longer addressed by the DP master or the IO controller. The output terminals of deactivated output modules carry zero or a substitute value. The process image of the input of deactivated input modules is set to "0".

A deactivated station can be removed from the bus without an error message; it is not signaled as faulty or missing. The calls of the asynchronous error organization blocks OB 85 (program execution error, if the user data of the deactivated station is present in an automatically updated process image) and OB 86 (station failure) are suppressed. They must no longer address the station from the program following deactivation, otherwise an I/O access error with calling of OB 122 will result in the case of direct access, or the station will be signaled as not being present when reading the data record with SFC 59 RD_REC or SFB 52 RDREC.

Use SFC 12 D_ACT_DP to reactivate a deactivated station. The station is configured and parameterized by the DP master or IO controller as with the return of a station. When activating, the asynchronous error OBs 85 and 86 are not started. If the BUSY parameter has the signal state "0" following activation, the station can be addressed by the user program.

In the case of a cold or warm restart, the CPU's operating system automatically activates the deactivated stations. An S7-300 CPU does not start up until all stations have been activated. An S7-400 CPU starts up and reports I/O access errors until the stations have been activated. The station status is retained during a hot restart.

Triggering interrupts with PROFIBUS DP

You can additionally use the following system functions with PROFIBUS DP:

- ▷ SFB 75 SALRM
Initiate interrupt
- ▷ SFC 7 DP_PRAL
Initiate hardware interrupt

The parameters of the system blocks are listed in Table 20.12.

SFB 75 SALRM

Trigger alarm

With SFB 75 SALRM, you initiate a diagnostic interrupt or process interrupt in the DP master associated with an intelligent slave from the user program of that slave. You define the type of interrupt using the ATYPE parameter.

The interrupt request is initiated with REQ = "1"; the DONE, BUSY, ERROR and STATUS parameters indicate the job status. The job is complete (BUSY = "0") when the interrupt OB in the DP master has been executed.

The transfer memory between the DP master and the intelligent DP slave can be divided into individual address areas that represent individual modules from the viewpoint of the master CPU. You can initiate an interrupt in the master

Tabelle 20.12 Parameters of the system blocks for triggering an interrupt with PROFIBUS DP

Block	Parameter	Declaration	Data type	Assignment, description
SFB 75	REQ	INPUT	BOOL	Request for triggering if REQ = "1"
	ID	INPUT	DWORD	Address of an address area in the transfer memory
	ATYPE	INPUT	INT	Interrupt type: 1 = diagnostics interrupt 2 = hardware interrupt
	ASPEC	INPUT	INT	Interrupt ID: 0 = no additional information 1 = slot faulty (UP) 2 = slot no longer faulty (DOWN) 3 = slot still faulty (DOWN)
	LEN	INPUT	INT	Length (in bytes) of additional interrupt information to be sent (max. 16)
	DONE	OUTPUT	BOOL	Interrupt has been transmitted if DONE = "1"
	BUSY	OUTPUT	BOOL	Interrupt transfer still running if BUSY = "1"
	ERROR	OUTPUT	BOOL	Error occurred if ERROR = "1"
	STATUS	OUTPUT	DWORD	Error information
SFC 7	AINFO	IN_OUT	ANY	Source area for the additional interrupt information
	REQ	INPUT	BOOL	Request for triggering if REQ = "1"
	IOID	INPUT	BYTE	B#16#54 = input ID B#16#55 = output ID
	LADDR	INPUT	WORD	Start address of an address area in the transfer memory
	AL_INFO	INPUT	DWORD	Interrupt ID (transfer to the start information of the interrupt OB)
	RET_VAL	RETURN	INT	Error information
	BUSY	OUTPUT	BOOL	Still no acknowledgment from DP master if BUSY = "1"

for each of these address areas ("virtual" modules). You specify the address area with the ID parameter which you occupy with a user data address from the viewpoint of the slave CPU. Bit 15 contains the I/O identification: "0" corresponds to an input address, "1" to an output address. The start information of the interrupt OB then contains the addresses of the "modules" initiating the interrupt from the viewpoint of the master CPU.

Use the AINFO parameter to transfer supplementary interrupt information which you have defined and which can be evaluated in the interrupt OB of the master CPU. The reference to AINFO is as an ANY pointer to a data area. The length of the sent information is defined by the LEN parameter and by the area length of the ANY pointer (the shorter length is decisive).

The first 4 bytes are displayed in the start information of the interrupt OB in the master CPU in bytes 8 to 11 (the variable OBxx_POINT_ADDR for process interrupts, the data record DS 0 for the diagnostic interrupt). The complete supplementary interrupt information can be read in the master CPU using SFB 54 RALRM.

SFC 7 DP_PRAL

Initiate a process interrupt

With SFC 7 DP_PRAL, you initiate a process interrupt in the DP master associated with an intelligent slave from the user program of that slave.

At the parameter AL_INFO you transfer an interrupt ID defined by you that is transferred to the start information of the interrupt OB called

in the DP master (variable `OBxx_POINT_ADDR`). The interrupt request is initiated with `REQ = "1"`; the parameters `RET_VAL` and `BUSY` indicate the job status. The job is complete when the interrupt `OB` in the DP master has been executed.

The transfer memory between the DP master and the intelligent DP slave can be divided into individual address areas that represent individual modules from the viewpoint of the master CPU. The lowest address of an address area is taken as the module starting address. You can initiate a process interrupt in the master for each of these address areas ("virtual" modules).

You specify an address area at SFC 7 with the parameters `IOID` and `LADDR` from the viewpoint of the slave CPU (the I/O ID and the starting address of the slave side). The start information of the interrupt `OB` then contains the addresses of the "module" initiating the interrupt from the viewpoint of the master CPU.

System blocks for PROFIBUS DP

You can additionally use the following system functions with PROFIBUS DP:

- ▷ SFC 11 `DPSYN_FR`
Send SYNC/FREEZE commands
- ▷ SFC 13 `DPNRM_DG`
Read diagnostic data from a DP standard slave
- ▷ SFC 103 `DP_TOPOL`
Determine bus topology

The parameters of these system functions are shown in Table 20.13.

With DPV1 mode set and DP slaves which support the DPV1 functionality, you can use other system blocks to parameterize and read the diagnostic data (see Chapters 21.9.3, "Reading additional Interrupt Information" and 22.5, "Parameterizing Modules").

SFC 11 `DPSYC_FR`

Send SYNC/FREEZE commands

Using SFC 11 `DPSYC_FR` you send the SYNC, UNSYNC, FREEZE, and UNFREEZE commands to a SYNC/FREEZE group which you have configured in the hardware configura-

tion. The SEND is initiated with `REQ = "1"` and is completed when `BUSY = "0"` is signaled.

In the parameter GROUP each group occupies one bit (from bit 0 = group 1 to bit 7 = group 8). The commands in the parameter MODE are also organized by bit:

- ▷ UNFREEZE, if bit 2 = "1"
- ▷ FREEZE, if bit 3 = "1"
- ▷ UNSYNC, if bit 4 = "1"
- ▷ SYNC, if bit 5 = "1"

A SYNC and an UNSYNC command or a FREEZE and an UNFREEZE command must not be triggered simultaneously in a call.

In this way, SYNC mode and FREEZE mode on the DP slaves are first switched off. The inputs of the DP slaves are scanned in sequence by the DP master and the outputs of the DP slaves are modified; the DP slaves pass the received output signals immediately to the output terminals.

If you want to "freeze" the input signals of several DP slaves at a specific time, you output the command FREEZE to the relevant group. The input signals then read in sequence by the DP master have the signal states they had when "frozen". These input signals retain their value until you output another FREEZE command to cause the DP slaves to read in and hold the current input signals, or until you switch the DP slaves back to the "normal" mode with the UNFREEZE command.

If you want to output the output signals of several DP slaves synchronously at a specific time, first output the SYNC command to the relevant group. The addressed DP slaves then hold the current signals at the output terminals. Now you can transfer the desired signal states to the DP slaves. Following the transfer, you output the SYNC command again; this causes the DP slaves to switch the received output signals simultaneously through to the output terminals. The accessed DP slaves then hold the signals at the output terminals until you switch through the new output signals with a new SYNC command, or until you switch the DP slaves back to their "normal" mode with the UNSYNC command.

Table 20.13 Parameters of the SFCs for addressing the distributed I/O

SFC	Parameter	Declaration	Data type	Assignment, description
11	REQ	INPUT	BOOL	Request for sending if REQ = "1"
	LADDR	INPUT	WORD	Configured diagnostics address of the DP master
	GROUP	INPUT	BYTE	DP slave group (from the hardware configuration)
	MODE	INPUT	BYTE	Command (see text)
	RET_VAL	RETURN	INT	Error information
	BUSY	OUTPUT	BOOL	Job still running if BUSY = "1"
13	REQ	INPUT	BOOL	Request for reading if REQ = "1"
	LADDR	INPUT	WORD	Configured diagnostics address of the DP slave
	RET_VAL	RETURN	INT	Error information
	RECORD	OUTPUT	ANY	Destination area for the read diagnostic data
	BUSY	OUTPUT	BOOL	Read operation still running if BUSY = "1"
103	REQ	INPUT	BOOL	Triggering of topology determination if REQ = "1"
	R	INPUT	BOOL	Cancellation of topology determination if R = "1"
	DP_ID	INPUT	INT	ID of the DP master system whose topology is to be determined
	RET_VAL	RETURN	INT	Error information for the SFC
	BUSY	OUTPUT	BOOL	Determination still running if BUSY = "1"
	DPR	OUTPUT	BYTE	PROFIBUS address of the diagnostics repeater reporting the error
	DPRI	OUTPUT	BYTE	Measuring segment and error information of the diagnostics repeater reporting the error

SFC 13 DPNRM_DG *Read diagnostic data*

SFC 13 DPNRM_DG reads diagnostic data from a DP slave. The read procedure is initiated with REQ = "1", and is terminated when BUSY = "0" is returned. Function value RET_VAL then contains the number of bytes read. Depending on the slave, diagnostic data may comprise from 6 to 240 bytes. If there are more than 240 bytes, the first 240 bytes are transferred and the relevant overflow bit is then set in the data.

The parameter RECORD writes to the area in which the read data are stored. Variables of data type ARRAY and STRUCT or an ANY pointer of data type BYTE (e.g. P#DBzDBXy xBY-TEnnn) are permissible as actual parameters.

Please note that the SFC 13 DPMRM_DG is an asynchronous system function. It must be processed until the BUSY parameter signals the status "0". The SFB 54 RALRM is available with newer CPUs which provides the data in

synchronous mode, i.e. immediately following the call.

SFC 103 DP_TOPOL *Determine bus topology*

SFC 103 DP_TOPOL uses diagnostics repeaters to determine the bus topology of the DP-master system whose ID you specify in the DP_ID parameter. Determination is triggered by REQ = "1" and has been completed when BUSY = "0". You can abort determination of the topology using R = "1".

If an error which prevents determination of the bus topology is signaled by a diagnostics repeater, it is shown in the DPR and DPRI parameters. If several diagnostics repeaters signal errors, the error message of the first one is displayed, and the complete diagnostics information can be read with SFC 13 DPNRM_DG.

The error information in the DPRI parameter is differentiated between temporary and permanent faults. Temporary faults, such as a loose

contact, may be difficult to identify and may disappear on their own. You must eliminate permanent faults before you call the SFC 103 DP_TOPOL again to determine the topology.

After calling the SFC 103 DP_TOPOL, the determined data are available on the diagnostics repeater and can be read with the SFC 59 RD_REC or SFB 52 RDREC. The data comprise the topology of the bus segment (nodes and cable lengths), the contents of the segment diagnostics buffer (last ten events with error information, location and cause) and the statis-

tics data (information on the quality of the bus system).

System block for PROFINET

The following system function block sets the IP configuration during runtime:

- ▷ SFB 104 IP_CONF
Setting the IP configuration

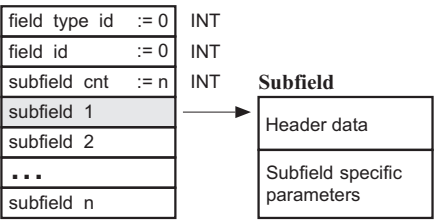
The parameters of this block are shown in Table 20.14.

Table 20.14 Parameters of the SFB for setting the IP configuration

SFB	Parameter	Declaration	Data type	Assignment, description
104	REQ	INPUT	BOOL	Request for setting if REQ = "1"
	LADDR	INPUT	WORD	Diagnostics address of the PROFINET interface
	CONF_DB	INPUT	ANY	Pointer to the configuration data
	DONE	OUTPUT	BOOL	Job completed without error if "1"
	BUSY	OUTPUT	BOOL	Job still running if BUSY = "1"
	ERROR	OUTPUT	BOOL	An error has occurred if "1"
	STATUS	OUTPUT	DWORD	Error information
	ERR_LOC	OUTPUT	DWORD	Error source

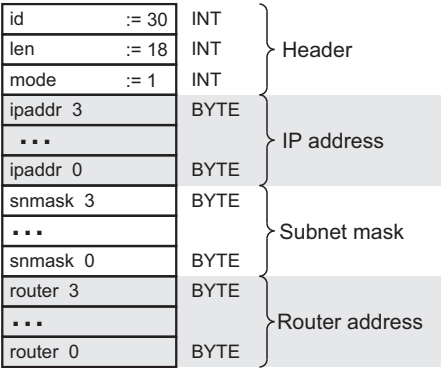
Data area CONF_DB

The data area CONF_DB consists of a header and several subfields. The subfields are currently defined for the IP parameters and the device name.



- *) If the length for the device name field is defined shorter, the length must be adapted in the header.
- If the device name is shorter than the field, the byte following the device name must be occupied by B#16#: 00.
- If B#16#: 00 is present at the first position of the device name, the name is deleted.

Subfield for the IP parameters



Subfield for the device name

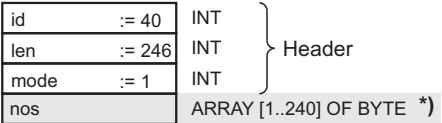


Fig. 20.27 Data structure for the IP configuration

SFB 104 IP_CONF

Setting the IP configuration

SFB 104 IP_CONF overwrites the IP address, the subnet mask, the router address, and – if the station is an IO device – the device name during runtime.

The job is triggered with REQ = "1"; the DONE, BUSY, ERROR, and STATUS parameters indicate the job status. The job is finished when BUSY = "0".

Changing the IP configuration during runtime must already be prepared in the hardware configuration: In the properties of the PN interface, click on the "Properties" button and check the "Obtain IP address in different manner" on the "Parameters" tab in the displayed window.

The ANY pointer in the CONF_DB parameter points to a data area which contains the new values for the IP configuration. It consists of a header containing the field type (= 0), the field ID (= 0), and the number of following subfields. The header is followed by the subfields.

At the moment, subfields are defined for the IP parameters and the device name (Figure 20.27).

20.5 Global Data Communication

20.5.1 Fundamentals

Global data communication (GD communication) is a communications service integrated into the operating system of the CPU and used for exchanging small volumes of non-time-critical data via the MPI bus. The transferable global data include

- ▷ Inputs and outputs (process images)
- ▷ Memory bits
- ▷ Data in data blocks
- ▷ Timer and counter values as data to be sent.

It is a requirement that the CPUs are networked together via the MPI interface or connected via the K bus as in the S7-400 mounting rack. All CPUs must exist in the same STEP 7 project in order to be able to configure GD communication.

The cyclic GD communication service does not require an operating system: there are system functions available for event-driven GD communication on the S7-400.

Please note that a receiver CPU does not acknowledge receipt of global data. The sender

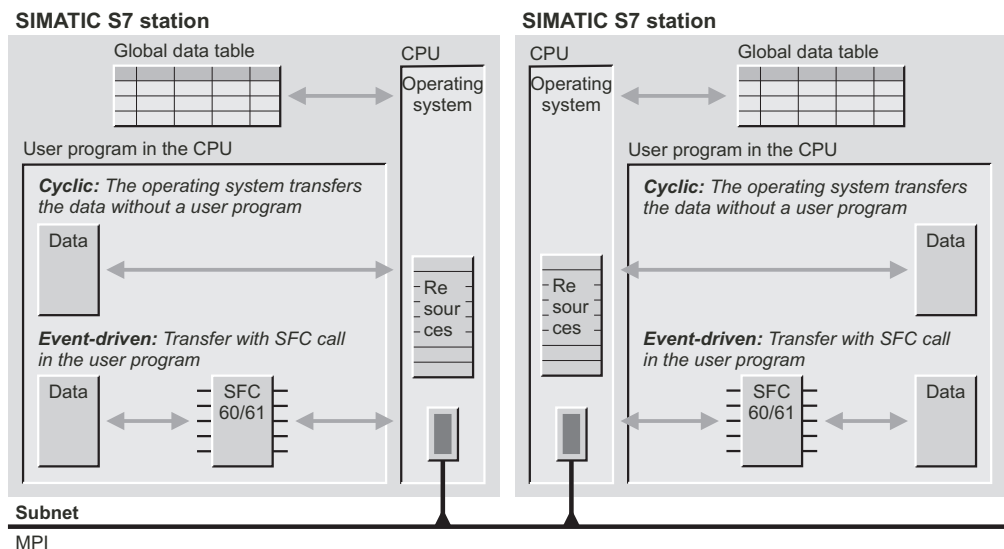


Figure 20.28 Global Data Communication

therefore does not receive any response to tell it if a receiver has received data and if so, which one. However, you can screen the communication status between two CPUs as well as the overall status of all GD circles of a CPU.

Sending and receiving global data is controlled with what are known as scan rates. These specify the number of (user program) cycles after which the CPU sends or receives the data. Sending and receiving takes place synchronously between the sender and the receiver at the cycle control point in each case, i.e. following cyclic program execution and before a new program cycle begins (like process image updating, for example).

Data is exchanged in the form of data packets (GD packets) between CPUs grouped into GD circles.

GD circle

The CPUs that exchange a shared GD packet form a GD circle. A GD circle can be any of the following

- ▷ The one-way connection of a CPU that sends a GD packet to several other CPUs that then receive that packet.
- ▷ The two-way connection between two CPUs where each of the two CPUs can send a GD packet to the other.
- ▷ The two-way connection between three CPUs where each of the three CPUs can send one GD packet to the other two CPUs (S7-400 CPUs only).

Up to 15 CPUs can exchange data with each other in one GD circle. One CPU can also belong to several GD circles. See Table 20.15 for the resources of each individual CPU here.

GD packet

A GD packet comprises the packet header and one or more global data elements (GD elements):

- ▷ Packet header (8 bytes)
- ▷ Identification of 1st GD element (2 bytes)
- ▷ User data of 1st GD element (x bytes)
- ▷ Identification of 2nd GD element (2 bytes)
- ▷ User data of 2nd GD element (x bytes)
- etc.

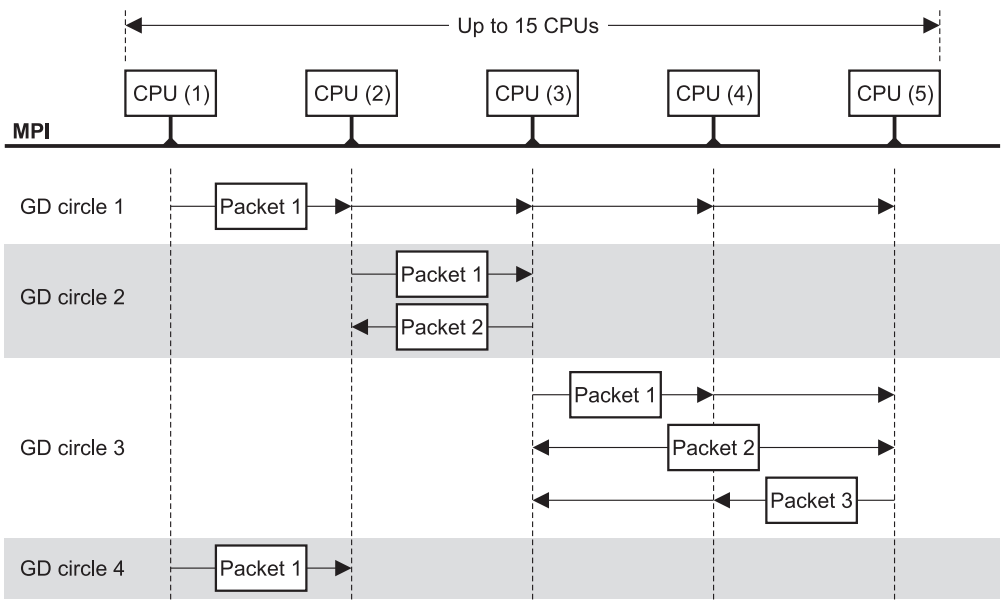


Figure 20.29 Example of GD Circles

Table 20.15 CPU Resources for Global Data Communication

GD resources	CPU 312 CPU 313 CPU 314 CPU3xxC	CPU 315 CPU 317	CPU 412 CPU 414	CPU 416 CPU 417
Max. number of:				
GD circles per CPU	4	8	8	16
Receive GD packets per CPU	4	8	16	32
Receive GD packets per circle	1	1	2	2
Send GD packets per CPU	4	8	8	16
Send GD packets per circle	1	1	1	1
Max. size of a GD packet	22 bytes	22 bytes	64 bytes	64 bytes
Max. data consistency	22 bytes	22 bytes	1 variable	1 variable

Each GD element consists of 2 bytes of description and the actual net data. 3 bytes are required in the GD packet to transfer a memory byte, 4 bytes are required for a memory word, and 6 bytes for a memory doubleword. A Boolean variable occupies 1 byte of net data; it therefore requires the same space as a byte-sized variable. Timer and counter values with 2 bytes each occupy 4 bytes each in the GD packet.

A GD element can also be an address area. MB 0:15, for example, represents the area from memory byte MB 0 to MB 15, and DB20.DBW14:8 represents the data area located in DB 20 that starts from data word DBW 14 and comprises 8 data words.

The maximum size of a GD packet is 32 bytes on the S7-300 and 64 bytes on the S7-400. The maximum number of net data bytes per packet is reached with the transfer of only one GD element that contains up to 22 bytes on the S7-300 and up to 54 bytes on the S7-400.

Data consistency

The data consistency covers one GD element. If a GD element overwrites a CPU-specific variable, the areas specified in Table 20.15 apply.

If a GD element is greater than the length of the data consistency, blocks with consistent data of the relevant length are formed, starting with the first byte.

20.5.2 Configuring GD communication

Requirements

You must have created a project, there must be an MPI subnetwork available and you must have configured the S7 stations. The CPU, at least, must be available in the stations. Under the “Properties” button of the MPI interface, on the “General” tab of the properties window of the CPU (double-click on the CPU line in the Hardware Configuration or on the line with the MPI interface submodule), you can set the MPI address and select the MPI subnetwork with which the CPU is connected.

Global data table

You configure GD communication by filling out a table. With the icon for the MPI subnetwork selected in the SIMATIC Manager or in the network configuration, you can call up an empty global data table with Options → Define global data. Select the station in the left half of the project selection window that then opens, and select the CPU in the right half. This CPU is accepted into the global data table with “OK”.

Proceed in exactly the same way with the other CPUs participating in GD communication. A global data table can contain up to 15 CPU columns.

To configure data transfer between CPUs, select the first line under the send CPU and specify the address whose value is to be transferred (terminate with RETURN).

Table 20.16 Example of a GD Table with Status and Scan Rates

GD Identifier	Station 417 \ CPU417 (3)	Station 417 \ CPU414 (4)	Station 416 \ CPU 416 (5)	Station 315 Slave \ CPU315 (7)	Station 314CP \ CPU314 (10)
GST	MD100	MD100	MD100	DB10.DBD200	DB10.DBD200
GDS 1.1	DB9.DBD0		MD92	DB10.DBD204	DB10.DBD204
SR 1.1	44	0	44	8	8
GD 1.1.1	>DB9.DBW10		MW90	DB10.DBW208	DB10.DBW208
GDS 2.1	MD96	MD96			
SR 2.1	44	23	0	0	0
GD 2.1.1	>Z10:10	DB19.DBW20:10			
GDS 3.1			MD96		
SR 3.1	0	0	44	8	8
GD 3.1.1			>MW98	DB10.DBW220	DB10.DBW210

With EDIT → SENDER, you define this value as the value to be sent, indicated by a prefixed character “>” and shading. In the same line under Receiver CPU, you enter the address that is to accept the value (the property “Receiver” is set as default). You may use timer and counter functions only as senders; the receiver must be an address of word width for each timer or counter function.

A line can contain several receivers but only one sender (Table 20.16). After filling this in, you select GD TABLE → COMPILE.

After compiling (phase 1), the system data created are sufficient for global data communication. If you also configure the GD status (the status of the GD connection) and the scan rates, you must then compile the GD table for a second time.

GD ID

Following error-free compiling, STEP 7 completes the “GD ID” column. The GD ID shows you how the transferred data are structured into GD circles, GD packets and GD elements. For example, the GD ID “GD 2.1.3” corresponds to GD circle 2, GD packet 1, GD element 3. You can then find the resource assignment (number of GD circles) per CPU in the CPU column of the global data table.

GD status

Following compiling, you can enter the addresses for the communication status into the

global data table with VIEW → GD STATUS. The overall status (GST) shows the status of all communications connections in the table. The status (GDS) shows the status of a communications connection (a transmitted GD packet). The status is shown in a doubleword in each case.

Scan rates

The GD communication service requires a significant portion of execution time in the CPU operating system and demands transmission time on the MPI bus. To keep this “communications load” to a minimum, it is possible to specify a “scan rate”. A scan rate specifies the number of program cycles after which the data (or more precisely, a GD packet) are to be sent or received.

Since the data are not updated in every program cycle with a scan rate, you should avoid sending time-critical data via this form of communication.

After the first (error-free) compilation, you can use VIEW → SCAN RATES to define the scan rates (SRs) yourself for each GD packet and each CPU. The scan rate is set as standard in such a way that with an “empty” CPU (no user program) the GD packets are sent and received approximately every 10 ms. If a user program is then loaded, the time interval increases.

You can enter the scan rates in the area between 1 and 255. Please note, that as the scan rates

Table 20.17 SFC Parameters for GD Communication

Parameter	Present in SFC		Declaration	Data Type	Contents, Description
CIRCLE_ID	60	61	INPUT	BYTE	Number of the GD circle
BLOCK_ID	60	61	INPUT	BYTE	Number of the GD packet to be sent or to be received
RET_VAL	60	61	RETURN	INT	Error information

decrease, the communications load on the CPU increases. To keep the communications load within tolerable limits, set the scan rate in the send CPU in such a way that the product of scan rate and cycle time on the S7-300 is greater than 60 ms and on the S7-400 greater than 10 ms. In the receive CPU, this product must be less than that in the send CPU to avoid the loss of any GD packets.

With scan rate 0, you switch off data exchange of the relevant GD packet if you only want to send or receive it event-driven with SFCs.

After configuring the GD status and the scan rates, you must compile the GD table for a second time. Then STEP 7 enters the compiled data in the *System data* object. GD communication becomes effective when you transfer the GD table to the connected CPUs with PLC → DOWNLOAD TO MODULE.

GD communication also becomes effective when the *System data* object, that contains all hardware settings and parameter settings, is transferred.

20.5.3 System Functions for GD Communication

In S7-400 systems, you can also control GD communication in your program. Additionally or alternatively to the cyclic transfer of global data, you can send or receive a GD packet with the following SFCs:

- ▷ SFC 60 GD_SND
Send GD packet
- ▷ SFC 61 GD_RCV
Receive GD packet

The parameters for these SFCs are listed in Table 20.17. The prerequisite for the use of these SFCs is a configured global data table.

After compiling this table, STEP 7 shows you, in the “GD Identifier” column, the numbers of the GD circles and GD packets which you need for parameter assignments.

SFC 60 GD_SND enters the GD packet in the system memory of the CPU and initiates transfer; SFC 61 GD_RCV fetches the GD packet from the system memory. If a scan rate greater than 0 has been specified for the GD packet in the GD table, cyclic transfer also takes place.

If you want to ensure data consistency for the entire GD packet when transferring with SFCs 60 and 61, you must disable or delay higher-priority interrupts and asynchronous errors on both the Send and Receive side during processing of SFC 60 or SFC 61.

The SFCs need not be called in pairs; “mixed” operation is also possible. For example, you can use SFC 60 GD_SND to have event-driven transmission of GD packets but then receive cyclically.

20.6 S7 Basic Communication

20.6.1 Station-Internal S7 Basic Communication

Fundamentals

With station-internal S7 basic communication, you can exchange data between programmable modules within a SIMATIC station. The communication functions required here are SFCs in the operating system of the CPU. These SFCs establish the communication connections themselves, if necessary. For this reason, these station-internal connections are not configured via the connection table (“Communication via non-configured connections”, SFC communication).

Station-internal S7 basic communication can take place, for example, parallel to cyclic data

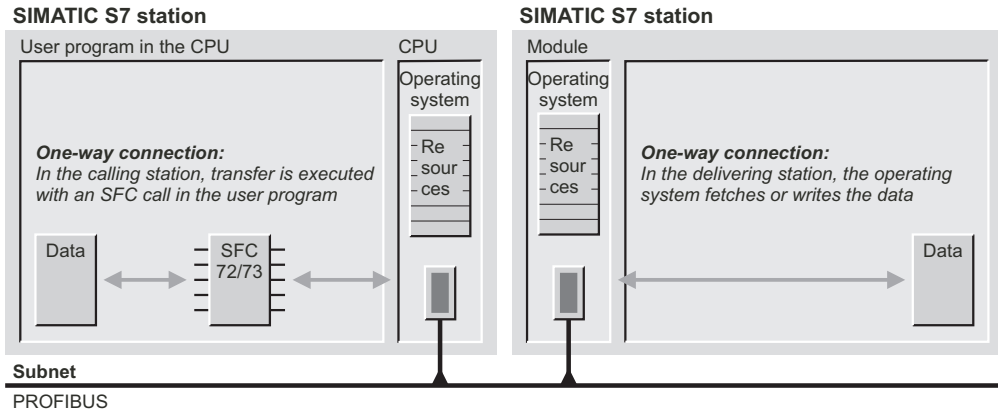


Figure 20.30 Station-Internal S7 Basic Communication

exchange via PROFIBUS DP between the master CPU and the slave CPU, with data transfer being event-driven (Figure 20.30).

Addressing the nodes, connections

Node identification is derived from the I/O address: at the LADDR parameter, you specify the module starting address and at the IOID parameter, you specify whether this address is in the input area or the output area.

These system functions establish the necessary communication connections dynamically and they terminate the connections following completion of the job (programmable). If a connection buildup cannot be executed due to lack of resources either in the sending device or in the receiving device, “Temporary lack of resources” is signaled. Transfer must then be reinitiated. There can only be one connection between two communication partners in each direction.

You can use one system function for different communication connections by modifying the block parameters at runtime. An SFC cannot interrupt itself. A program section in which one of these SFCs is used can only be modified in the STOP mode; following this, a cold or warm restart must then be executed.

User data, data consistency

These SFCs transfer up to 76 bytes as user data. Regardless of the direction of transfer, the oper-

ating system of a CPU arranges the user data in blocks that are consistent within themselves. The length of consistent transferred data is a CPU-specific variable. If two CPUs exchange data, the block size of the “passive” CPU is decisive for data consistency.

Configuring station-internal S7 basic communication

Station-internal S7 basic communication is a special case in that it requires no configuring since data transfer is handled via dynamic connections. You simply use an existing PROFIBUS subnetwork or you create one either in the SIMATIC Manager (select the *Project* object and then INSERT → SUBNETWORK → PROFIBUS) or in the Network Configuration (see Chapter 2.4, “Configuring the Network”).

Example: you have configured distributed I/O with a CPU 315-2DP as master. You use another CPU 315-2DP as an “intelligent” slave. You can now use station-internal S7 basic communication from both controllers to read and write data.

20.6.2 System Functions for Station-Internal S7 Basic Communication

The following system functions handle data transfers between two CPUs in the same station:

- ▷ SFC 72 I_GET
Read data

Table 20.18 SFC Parameters for Station-Internal S7 Basic Communication

Parameter	Present in SFC			Declaration	Data Type	Contents, Description
REQ	72	73	74	INPUT	BOOL	Initiate job with REQ = “1”
CONT	72	73	-	INPUT	BOOL	CONT = “1”: Connection remains intact after job terminates
IOID	72	73	74	INPUT	BYTE	B#16#54 = Input area, B#16#55 = Output area
LADDR	72	73	74	INPUT	WORD	Module start address
VAR_ADDR	72	73	-	INPUT	ANY	Data area in partner CPU
SD	-	73	-	INPUT	ANY	Data area in own CPU which contains the Send data
RET_VAL	72	73	74	RETURN	INT	Error information
BUSY	72	73	74	OUTPUT	BOOL	Job in progress when BUSY = “1”
RD	72	-	-	OUTPUT	ANY	Data area in own CPU which will take the Receive data

▷ SFC 73 I_PUT
Write data

▷ SFC 74 I_ABORT
Disconnect

The parameters for these SFCs are listed in Table 20.18.

SFC 72 I_GET **Read data**

A job is initiated with REQ = “1” and BUSY = “0” (“first call”). While the job is in progress, BUSY is set to “1”. Changes to the REQ parameter no longer have any effect. When the job is completed, BUSY is reset to “0”. If REQ is still “1”, the job is immediately restarted.

When the read procedure has been initiated, the operating system in the partner CPU assembles and sends the requested data. An SFC call transfers the Receive data to the destination area. RET_VAL then shows the number of bytes transferred.

If CONT is = “0”, the communication link is broken. If CONT is = “1”, the link is maintained. The data are also read when the communication partner is in STOP mode.

The RD and VAR_ADDR parameters describe the area from which the data to be transferred are to be read or to which the receive data are to be written. Actual parameters may be addresses, variables or data areas addressed

with an ANY pointer. The Send and Receive data are not checked for identical data types.

SFC 73 I_PUT **Write data**

A job is initiated with REQ = “1” and BUSY = “0” (“first call”). While the job is in progress, BUSY is set to “1”. Changes to the REQ parameter no longer have any effect. When the job is completed, BUSY is reset to “0”. If REQ is still “1”, the job is immediately restarted.

When the write procedure has been initiated, the operating system transfers all data from the source area to an internal buffer on the first call, and sends them to the partner in the link. There, the receiver writes the data into data area VAR_ADDR. BUSY is then set to “0”. The data are also written when the receiving partner is at STOP.

The SD and VAR_ADDR parameters describe the area from which the data to be transferred are to be read or to which the receive data are to be written. Actual parameters may be addresses, variables or data areas addressed with an ANY pointer. The Send and Receive data are not checked for identical data types.

SFC 74 I_ABORT **Disconnect**

REQ = “1” breaks a connection to the specified communication partner. With I_ABORT, you

can break only those connections established in the same station with I_GET or I_PUT.

While the job is in progress, BUSY is set to “1”. Changes to the REQ parameter no longer have any effect. When the job is completed, BUSY is reset to “0”. If REQ is still “1”, the job is immediately restarted.

20.6.3 Station-External S7 Basic Communication

Fundamentals

With station-external S7 basic communication, you can have event-driven data exchange between SIMATIC S7 stations. The stations must be connected to each other via an MPI subnetwork. The communications functions required for this are SFCs in the operating system of the CPU. These SFCs establish the communication connections themselves, if necessary. For this reason, these station-external connections are not configured via the connection table (“Communication via non-configured connections”).

Station-external S7 basic communication can execute event-driven data transfer, for example, parallel to cyclic global data communication.

Addressing the nodes, connections

These functions address nodes that are on the same MPI subnetwork. The node identification is derived from the MPI address (DEST_ID parameter).

These system functions set up the required communication links dynamically and – if specified – break them when the job has been executed. If a connection cannot be established because of a lack of resources in either the sender or the receiver, “temporary lack of resources” is reported. The transfer must then be retried. Between two communication partners, there can be only one connection in each direction.

On a transition from RUN to STOP, all active connections (all SFCs except X_RECV) are cleared.

By modifying the block parameters at run time, you can utilize a system function for different communication links. An SFC may not interrupt itself. You may modify a program section in which one of these SFCs is used only in STOP mode; a cold or warm restart must then be executed.

User data, data consistency

These SFCs transfer a maximum of 76 bytes of user data. A CPU’s operating system combines the user data into blocks consistent within

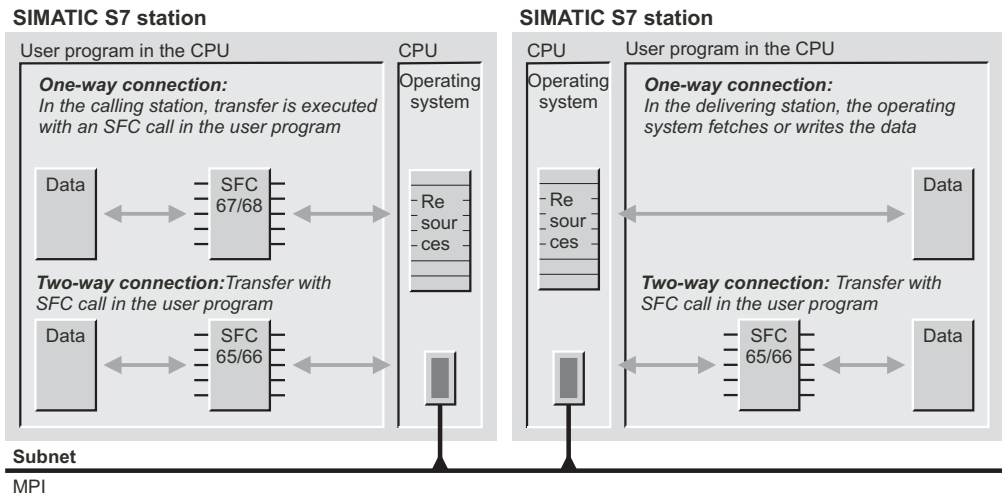


Figure 20.31 Station-External S7 Basic Communication

themselves, without regard to the direction of transfer. The length of consistent transferred data is a CPU-specific variable.

If two CPUs exchange data via X_GET or X_PUT, the block size of the “passive” CPU is decisive to data consistency of the transferred data.

In the case of a SEND/RECEIVE connection, all data of a call are consistent.

Configuring station-external S7 basic communication

Station-external S7 basic communication is a special case in that it requires no configuring since data transfer is handled via dynamic connections. You simply use an existing MPI sub-network or you create one.

Example: you have a divided S7-400 mounting rack with one CPU 416 in each section. In addition, you connect an S7-300 station with a CPU 314 via an MPI cable to one of the S7-400s. You configure all three CPUs in the Hardware Configuration, for example, as “networked” via

an MPI subnetwork. You can now use station-external S7 basic communication from all three controllers to exchange data.

20.6.4 System Functions for Station-External S7 Basic Communication

The following system functions handle data transfers between partners in different stations:

- ▷ SFC 65 X_SEND
Send data
- ▷ SFC 66 X_RCV
Receive data
- ▷ SFC 67 X_GET
Read data
- ▷ SFC 68 X_PUT
Write data
- ▷ SFC 69 X_ABORT
Disconnect

The parameters for these SFCs are listed in Table 20.19.

Table 20.19 SFC Parameters for Station-External S7 Basic Communication

Parameter	Present in SFC					Declaration	Data Type	Contents, Description
REQ	65	-	67	68	69	INPUT	BOOL	Job initiation with REQ = “1”
CONT	65	-	67	68	-	INPUT	BOOL	CONT = “1”: Connection is maintained when job is completed
DEST_ID	65	-	67	68	69	INPUT	WORD	Partner's node identification (MPI address)
REQ_ID	65	-	-	-	-	INPUT	DWORD	Job identification
VAR_ADDR	-	-	67	68	-	INPUT	ANY	Data area in partner CPU
SD	65	-	-	68	-	INPUT	ANY	Data area in own CPU which contains the Send data
EN_DT	-	66	-	-	-	INPUT	BOOL	If “1”: Accept Receive data
RET_VAL	65	66	67	68	69	RETURN	INT	Error information
BUSY	65	-	67	68	69	OUTPUT	BOOL	Job in progress when BUSY = “1”
REQ_ID	-	66	-	-	-	OUTPUT	DWORD	Job identification
NDA	-	66	-	-	-	OUTPUT	BOOL	When “1”: Data received
RD	-	66	67	-	-	OUTPUT	ANY	Data area in own CPU which will accept the Receive data

SFC 65 X_SEND**Send data**

A job is initiated with REQ = "1" and BUSY = "0" ("first call"). While the job is in progress, BUSY is set to "1"; changes to the REQ parameter now no longer have any effect. When the job terminates, BUSY is set back to "0". If REQ is still "1", the job is immediately restarted.

On the first call, the operating system transfers all data from the source area to an internal buffer, then transfers the data to the partner CPU.

BUSY is "1" for the duration of the send procedure. When the partner has signaled that it has fetched the data, BUSY is set to "0" and the send job terminated.

If CONT is = "0", the connection is broken and the respective CPU resources are available to other communication links. If CONT is = "1", the connection is maintained. The REQ_ID parameter makes it possible for you to assign an ID to the Send data which you can evaluate with SFC X_RCV.

The SD parameter describes the area from which the data to be sent are to be read. Actual parameters may be addresses, variables, or data areas addressed with an ANY pointer. Send and Receive data are not checked for matching data types.

SFC 66 X_RCV**Receive data**

The Receive data are placed in an internal buffer. Multiple packets can be put in a queue in the chronological order of their arrival.

Use EN_DT = "0" to check whether or not data were received; if so, NDA is "1", RET_VAL shows the number of bytes of Receive data, and REQ_ID is the same as the corresponding parameter in SFC 65 X_SEND. When EN_DT is = "1", the SFC transfers the first (oldest) packet to the destination area; NDA is then "1" and RET_VAL shows the number of bytes transferred. If EN_DT is "1" but there are no data in the internal queue, NDA is "0".

On a cold or warm restart, all data packets in the queue are rejected.

In the event of a broken connection or a restart, the oldest entry in the queue, if already "queued" with EN_DT = "0", is retained; otherwise, it is rejected like the other queue entries.

The RD parameter describes the area to which the Receive data are to be written. Actual parameters may be addresses, variables, or data areas addressed with an ANY pointer.

Send and Receive data are not checked for matching data types. When the Receive data are irrelevant, a "blank" ANY pointer (NIL pointer) as RD parameter in X_RCV is permissible.

SFC 67 X_GET**Read data**

A job is initiated with REQ = "1" and BUSY = "0" ("first call"). While the job is in progress, BUSY is set to "1"; changes to the REQ parameter now no longer have any effect.

When the job terminates, BUSY is set back to "0". If REQ is still "1", the job is immediately restarted.

When the read procedure has been initiated, the operating system in the partner CPU assembles and sends the data required under VAR_ADDR. On an SFC call, the Receive data are entered in the destination area specified at the RD parameter. RET_VAL then shows the number of bytes transferred.

If CONT is "0", the communication link is broken. If CONT is "1", the connection is maintained. The data are then read even when the communication partner is in STOP mode.

The RD and VAR_ADDR parameters describe the area from which the data to be sent are to be read or to which the Receive data are to be written. Actual parameters may be addresses, variables, or data areas addressed with an ANY pointer. Send and Receive data are not checked for matching data types.

SFC 68 X_PUT**Write data**

A job is initiated with REQ = "1" and BUSY = "0" ("first call"). While the job is in progress, BUSY is set to "1"; changes to the REQ parameter now no longer have any effect.

When the job terminates, BUSY is set back to “0”. If REQ is still “1”, the job is immediately restarted.

When the write procedure has been initiated, the operating system transfers all data from the source area specified at the SD parameter to an internal buffer on the first call, then sends the data to the partner CPU. There, the partner CPU’s operating system writes the Receive data to the data area specified at the VAR_ADDR parameter. BUSY is then set to “0”.

The data are written even if the communication partner is in STOP mode.

The RD and VAR_ADDR parameters describe the area from which the data to be sent are to be read or to which the Receive data are to be written. Actual parameters may be addresses, variables, or data areas addressed with an ANY pointer. Send and Receive data are not checked for matching data types.

**SFC 69 X_ABORT
Disconnect**

REQ = “1” breaks an existing connection to the specified communication partner. The SFC X_ABORT can be used to break only those con-

nections established in the CPU’s own station with the SFCs X_SEND, X_GET or X_PUT.

20.7 S7 Communication

20.7.1 Fundamentals

With S7 communication, you transfer larger volumes of data between SIMATIC S7 stations. The stations are connected to each other via a subnetwork; this can be an MPI subnetwork, a PROFIBUS subnetwork or an Ethernet subnetwork. The communications connections are static; they are configured in the connection table (“Communication via configured connections”).

With the S7-400, the communications functions are system function blocks SFBs integrated in the operating system of the CPUs. The associated instance data block is located in the user memory. If you want to use S7 communication, copy the interface description of the SFBs from the *Standard Library* under *System Function Blocks* to the *Blocks* container, generate an instance data block for each call and call the SFB with the associated instance data block.

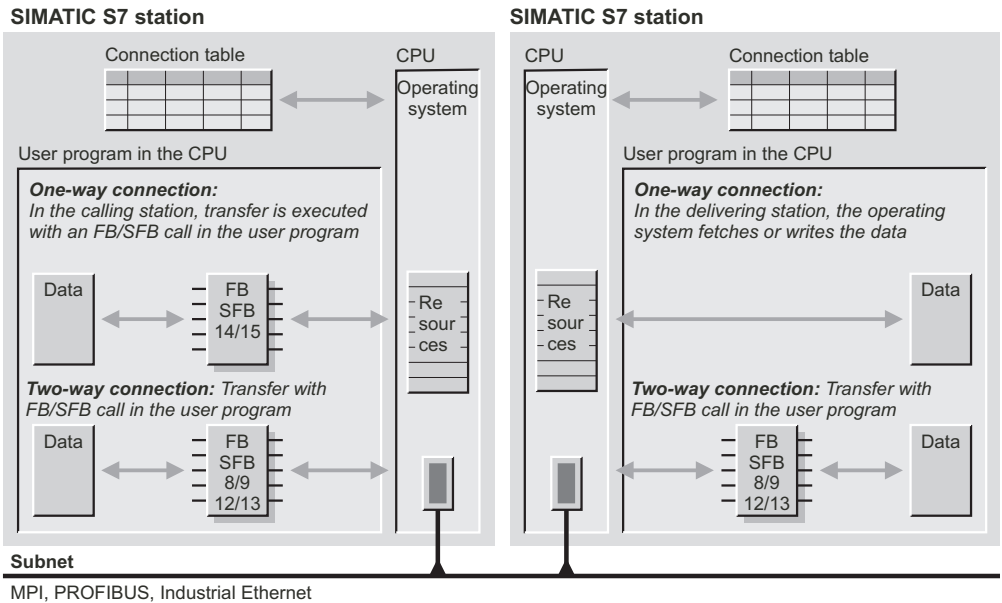


Figure 20.32 S7 Communication

With incremental input, you can also select the SFB from the program element catalog and have the instance data block generated automatically.

With the S7-300, the communication functions are standard function blocks FB which you can find in the *Standard Library* under *Communication Blocks*. Copy the function blocks you wish to use into the container *Blocks* and then use them just like “normal” function blocks.

Configuring S7 communication

The prerequisite for the S7 communication is a configured connection table in which the communication links are defined.

A communication link is specified by a connection ID for each communication partner. STEP 7 assigns the connection IDs when it compiles the connection table. Use the “local ID” to initialize the FB or SFB in the module from which the connection is viewed, and the “remote ID” to initialize the FB or SFB in the partner module.

The same logical connection can be used for different Send/Receive requests. To distinguish between them, you must add a job ID to the connection ID in order to define the relationship between the Send block and Receive block.

Initialization

S7 communication must be initialized at restart so that the connection to the communication partner can be established. Initialization takes place in the CPU that receives the attribute “Active connection buildup = Yes” in the connection table. You call the communication blocks used in cyclic operation in a restart OB and initialize the parameters (provided they are available) as follows:

- ▷ REQ = FALSE
- ▷ ID = local connection ID from the connection table (data type WORD W#16#xxxx)
- ▷ R_ID = job ID which you can design for a “pair of blocks” (data type DWORD DW#16#xxxx xxxx)

- ▷ PI_NAME = variable with the contents “P_PROGRAM” in ASCII coding (e.g. ARRAY[1..9] OF CHAR).

The blocks must continue to be called in a program loop until the DONE parameter has signal state “1”. The parameters ERROR and STATUS give information concerning the errors that have occurred and the job status.

You do not need to switch the data areas at startup (concerns the ADDR_n, RD_n, and SD_n parameters). Exception in S7-400 systems: with SFB 8 USEND, SFB 9 URCV, SFB 14 GET and SFB 15 PUT, the communication buffers for guaranteeing consistency are created when called for the first time, and these define the maximum data quantity per transfer for all further calls.

Cyclic operation

In cyclic operation, you call the communication blocks absolutely and you control data transfer via the parameters REQ and EN_R. You must evaluate the results in the NDR, DONE, ERROR and STATUS parameters immediately following each processing of a communication block since they are only valid up to the next call.

With the S7-300, parameters of data type ANY (SD_1, RD_1, ADDR_1) can only be supplied with flags and data as the address.

20.7.2 Two-Way Data Exchange

For two-way data exchange, you require one SEND block and one RECEIVE block each at the ends of a connection. Both blocks carry the connection IDs that are located in the connection table in the same line. You can also use several “block pairs” which are then distinguished from each other by the job ID.

The following SFBs are available for two-way data interchange:

- ▷ FB/SFB 8 USEND
Uncoordinated sending of a data packet of CPU-specific length
- ▷ FB/SFB 9 URCV
Uncoordinated receiving of a data packet of CPU-specific length

Table 20.20 FB/SFB Parameters for Sending and Receiving Data

Parameter	Present in FB/SFB				Declaration	Data Type	Contents, Description
REQ	8	-	12	-	INPUT	BOOL	Start data exchange
EN_R	-	9	-	13	INPUT	BOOL	Receive ready
R	-	-	12	-	INPUT	BOOL	Abort data exchange
ID	8	9	12	13	INPUT	WORD	Connection ID
R_ID	8	9	12	13	INPUT	DWORD	Job ID
DONE	8	-	12	-	OUTPUT	BOOL	Job terminated
NDR	-	9	-	13	OUTPUT	BOOL	New data fetched
ERROR	8	9	12	13	OUTPUT	BOOL	Error occurred
STATUS	8	9	12	13	OUTPUT	WORD	Job status
SD_1	8	-	12	-	IN_OUT	ANY	First Send area
SD_2 ¹⁾	8	-	-	-	IN_OUT	ANY	Second Send area
SD_3 ¹⁾	8	-	-	-	IN_OUT	ANY	Third Send area
SD_4 ¹⁾	8	-	-	-	IN_OUT	ANY	Fourth Send area
RD_1	-	9	-	13	IN_OUT	ANY	First Receive area
RD_2 ¹⁾	-	9	-	-	IN_OUT	ANY	Second Receive area
RD_3 ¹⁾	-	9	-	-	IN_OUT	ANY	Third Receive area
RD_4 ¹⁾	-	9	-	-	IN_OUT	ANY	Fourth Receive area
LEN	-	-	12	13	IN_OUT	WORD	Length of data block in bytes

¹⁾ Not with FB 8 or FB 9

- ▷ FB/SFB 12 BSEND
Sending of a data block of up to 32 or 64 KB in length
- ▷ FB/SFB 13 BRCV
Receiving of a data block of up to 32 or 64 KB in length

FB/SFB 8 and FB/SFB 9 or FB/SFB 12 and FB/SFB 13 must always be used as a pair.

The parameters for these SFBs are listed in Table 20.20.

FB 8 USEND and FB 9 URCV **SFB 8 USEND and SFB 9 URCV** **Uncoordinated sending and receiving**

The SD_n and RD_n parameters are used to specify the variable or the area you want to transfer. The send area SD_n must correspond to the respective receive area RD_n. Use the parameters without gaps, beginning with 1. No values need be specified for unneeded parameters (like an FB, not all SFB parameters need be assigned values).

A positive edge at the REQ (request) parameter starts the data exchange, a positive edge at the R (reset) parameter aborts it. A “1” in the EN_R (enable receive) parameter signals that the partner is ready to receive data, “0” can be used to abort a current job.

When the NDR parameter has assumed the value “1” following the data transfer, call the block again – this time with EN_R = “0” – to prevent the receive area from being overwritten by new data during the data evaluation.

Initialize the ID parameter with the connection ID, which STEP 7 enters in the connection table for both the local and the partner (the two IDs may differ). R_ID allows you to choose a specifiable but unique job ID which must be identical for the Send and Receive block. This allows several pairs of Send and Receive blocks to share a single logical connection (as each has a unique ID).

With the S7-400, the system function blocks accept the actual values of the ID and R_ID parameters into their instance data block on the

first call. The first call establishes the communication relationship (for this instance) until the next warm or cold start. With the S7-300, you can change the assignment of the ID and R_ID parameters following each completed job.

With signal state “1” in the DONE or NDR parameter, the block signals that the job terminated without error. An error, if any, is flagged in the ERROR parameter. A value other than zero in the STATUS parameter indicates either a warning (ERROR = “0”) or an error (ERROR = “1”).

FB 12 BSEND and FB 13 BRCV
SFB 12 BSEND and SFB 13 BRCV
Block-oriented sending and receiving

Specify a pointer to the first byte of the data area in parameter SD_n or RD_n (the length of this actual parameter determines the maximum size of the communication buffer when called for the first time, it is not evaluated with the further calls); the number of bytes of Send or Receive data is in the LEN parameter.

Up to 64 KB (32 KB with non-integrated interface) may be transferred; the data are transferred in blocks (sometimes called frames), and the transfer itself is asynchronous to the user program scan. The LEN parameter is updated following each received block.

A positive edge at the REQ (request) parameter starts the data exchange, a positive edge at the R (reset) parameter aborts it. A “1” in the EN_R (enable receive) parameter signals that the partner is ready to receive data, “0” can be used to abort a current job.

When the NDR parameter has assumed the value “1” following the data transfer, call the block again – this time with EN_R = “0” – to prevent the receive area from being overwritten by new data during the data evaluation.

Initialize the ID parameter with the connection ID, which STEP 7 enters in the connection table for both the local and the partner (the two IDs may differ). R_ID allows you to choose a specifiable but unique job ID which must be identical for the Send and Receive block. This allows several pairs of Send and Receive blocks to share a single logical connection (as each has a unique ID).

With the S7-400, the system function blocks accept the actual values of the ID and R_ID parameters into their instance data block on the first call. The first call establishes the communication relationship (for this instance) until the next warm or cold start. With the S7-300, you can change the assignment of the ID and R_ID parameters following each completed job.

With signal state “1” in the DONE or NDR parameter, the block signals that the job terminated without error. An error, if any, is flagged in the ERROR parameter. A value other than zero in the STATUS parameter indicates either a warning (ERROR = “0”) or an error (ERROR = “1”).

20.7.3 One-Way Data Exchange

In one-way data exchange, the communication block call is located in only one CPU. In the partner CPU, the operating system handles the necessary communication functions.

The following blocks are available for one-way data interchange

- ▷ FB 14 GET
 FB 34 GET_E
 SFB 14 GET
 Read data from a partner CPU
- ▷ FB 15 PUT
 FB 35 PUT_E
 SFB 15 PUT
 Write data to a partner CPU

Table 20.21 lists the parameters for these blocks.

The operating system in the partner CPU collects the data read with FB/SFB 14; the operating system in the partner CPU distributes the data written with FB/SFB 15. A Send or Receive (user) program in the partner CPU is not required. The partner CPU can provide the required communications services both at RUN and STOP. The size of the transferred consistent data blocks depends on the (server) CPU used.

A positive edge at parameter REQ (request) starts the data interchange. Set the ID parameter to the connection ID entered by STEP 7 in the connection table.

Table 20.21 /FB/SFB Parameters for Reading and Writing Data

Parameter	Present in FB/SFB		Declaration	Data Type	Contents, Description
REQ	14	15	INPUT	BOOL	Start data exchange
ID	14	15	INPUT	WORD	Connection ID
NDR	14	-	OUTPUT	BOOL	New data fetched
DONE	-	15	OUTPUT	BOOL	Job terminated
ERROR	14	15	OUTPUT	BOOL	Error occurred
STATUS	14	15	OUTPUT	WORD	Job status
ADDR_1	14	15	IN_OUT	ANY	First data area in partner CPU
ADDR_2 ¹⁾	14	15	IN_OUT	ANY	Second data area in partner CPU
ADDR_3 ¹⁾	14	15	IN_OUT	ANY	Third data area in partner CPU
ADDR_4 ¹⁾	14	15	IN_OUT	ANY	Fourth data area in partner CPU
RD_1	14	-	IN_OUT	ANY	First Receive area
RD_2 ¹⁾	14	-	IN_OUT	ANY	Second Receive area
RD_3 ¹⁾	14	-	IN_OUT	ANY	Third Receive area
RD_4 ¹⁾	14	-	IN_OUT	ANY	Fourth Receive area
SD_1	-	15	IN_OUT	ANY	First Send area
SD_2 ¹⁾	-	15	IN_OUT	ANY	Second Send area
SD_3 ¹⁾	-	15	IN_OUT	ANY	Third Send area
SD_4 ¹⁾	-	15	IN_OUT	ANY	Fourth Send area

¹⁾ Not with FB 14 or FB 15

With a “1” in the DONE or NDR parameter, the block signals that the job terminated without error. An error, if any, is flagged with a “1” in the ERROR parameter. A value other than zero in the STATUS parameter is indicative of either a warning (ERROR = “0”) or an error (ERROR = “1”). You must evaluate the DONE, NDR, ERROR and STATUS parameters after *every* block call.

Use the ADDR_n parameter to specify the variable or the area in the partner CPU from which you want to fetch or to which you want to send the data. The areas in ADDR_n must coincide with the areas specified in SD_n or RD_n. Use the parameters without gaps, beginning with 1. Unneeded parameters are not assigned (as in an FB, an SFB does not have to have values for all parameters).

20.7.4 Transferring Print Data

SFB 16 PRINT allows you to transfer a format description and data to a printer via a CP 441

communications processor. Table 20.22 lists the parameters for this SFB.

A positive edge at the REQ parameter starts the data exchange with the printer specified by the ID and PRN_NR parameters. The block signals an error-free transfer by setting DONE to “1”. An error, if any, is flagged by a “1” in the ERROR parameter. A value other than zero in the STATUS parameter is indicative of either a warning (ERROR = “0”) or an error (ERROR = “1”). You must evaluate the DONE, ERROR and STATUS parameters after *every* block call.

Enter the characters to be printed in STRING format in the FORMAT parameter. You can integrate as many as four format descriptions for variables in this string, defined in parameters SD_1 to SD_4. Use the parameters without gaps, beginning with 1; do not specify values for unneeded parameters. You can transfer up to 420 bytes (the sum of FORMAT and all variables) per print request.

Table 20.22 Parameters for SFB 16 PRINT

Parameter	Declaration	Data Type	Contents, Description
REQ	INPUT	BOOL	Start data exchange
ID	INPUT	WORD	Connection ID
DONE	OUTPUT	BOOL	Job terminated
ERROR	OUTPUT	BOOL	Error occurred
STATUS	OUTPUT	WORD	Job status
PRN_NR	IN_OUT	BYTE	Printer number
FORMAT	IN_OUT	STRING	Format description
SD_1	IN_OUT	ANY	First variable
SD_2	IN_OUT	ANY	Second variable
SD_3	IN_OUT	ANY	Third variable
SD_4	IN_OUT	ANY	Fourth variable

20.7.5 Control Functions

The following SFBs are available for controlling the communication partner:

- ▷ **SFB 19 START**
Execute a cold or warm restart in the partner controller
- ▷ **SFB 20 STOP**
Switch the partner controller to STOP
- ▷ **SFB 21 RESUME**
Execute a hot restart in the partner controller

These SFBs are for one-way data exchange; no user program is required in the partner device for this purpose. The parameters for them are listed in Table 20.23.

A positive edge at the REQ parameter starts the data exchange. Enter as ID parameter the connection ID which STEP 7 entered in the connection table.

With a “1” in the DONE parameter, the block signals that the job terminated without error. An error, if any, is flagged by a “1” in the ERROR parameter. A value other than zero in the STATUS parameter is indicative of either a warning (ERROR = “0”) or an error (ERROR = “1”). You must evaluate the DONE, ERROR and STATUS parameters after *every* block call.

Specify as PI_NAME an array variable with the contents “P_PROGRAM” (ARRAY [1..9] OF CHAR). If you do not assign the ARG parameter, a warm start is triggered in the partner con-

troller; if “C” is assigned to ARG, a cold start is triggered if permissible in the partner controller. The IO_STATE parameter is currently irrelevant, and need not be assigned a value.

SFB 19 START executes a cold or warm restart of the partner CPU. Prerequisite is that the partner CPU is at STOP and that the mode selector is positioned to either RUN or RUN-P.

SFB 20 STOP sets the partner CPU to STOP. Prerequisite for error-free execution of this job request is that the partner CPU is not at STOP when the request is submitted.

SFB 21 RESUME executes a hot restart of the partner CPU. Prerequisite is that the partner device is at STOP, that the mode switch is set to either RUN or RUN-P, and that a hot restart is permissible at this time.

20.7.6 Monitoring Functions

The following system blocks are available for monitoring functions

- ▷ **SFB 22 STATUS**
Check partner status
- ▷ **SFB 23 USTATUS**
Receive partner status
- ▷ **SFC 62 CONTROL**
Check status of an SFB instance
- ▷ **FC 62 C_CNTRL**
Check status of a connection

Table 20.23 SFB Parameters for Partner Controller

Parameter	Present in SFB			Declaration	Data Type	Contents, Description
REQ	19	20	21	INPUT	BOOL	Start data exchange
ID	19	20	21	INPUT	WORD	Connection ID
DONE	19	20	21	OUTPUT	BOOL	Job terminated
ERROR	19	20	21	OUTPUT	BOOL	Error occurred
STATUS	19	20	21	OUTPUT	WORD	Job status
PI_NAME	19	20	21	IN_OUT	ANY	Program name (P_PROGRAM)
ARG	19	-	21	IN_OUT	ANY	With "C" as the value, a cold start is triggered in the partner controller, if permissible
IO_STATE	19	20	21	IN_OUT	BYTE	Irrelevant

▷ SFC 87 C_DIAG

Determine connection status

The parameters of these blocks are described in Tables 20.24, 20.25 and 20.26.

The following applies for these system blocks: an error is indicated with "1" at the ERROR parameter. If the STATUS parameter has a value not equal to zero, this indicates either a warning (ERROR = "0") or an error (ERROR = "1").

SFB 22 STATUS

Check the status of the partner device

SFB 22 STATUS fetches the status of the partner CPU and displays it in the PHYS (physical status), LOG (logical status) and LOCAL (operating status if the partner is an S7 CPU) parameters.

A positive edge at the REQ (request) parameter starts the query. Enter as ID parameter the connection ID which STEP 7 entered in the connection table.

With a "1" in the NDR parameter, the block signals that the job terminated without error. You must evaluate the NDR, ERROR and STATUS parameters after *every* block call.

SFB 23 USTATUS

Receive the status of the partner device

SFB 23 USTATUS receives the status of the partner, which it sends, unrequested, in the event of a change. The device status is displayed in the PHYS, LOG and LOCAL parameters.

A "1" in the EN_R (enable receive) parameter signals that the partner is ready to receive data. Initialize the ID parameter with the connection ID, which STEP 7 enters in the connection table.

With a "1" in the NDR parameter, the block signals that the job terminated without error. You must evaluate the NDR, ERROR and STATUS parameters after *every* block call.

SFC 62 CONTROL

Check the status of a communication instance

In S7-400 systems, SFC 62 CONTROL determines the status of a communication instance and the associated connection in the local controller. Enter the SFB's instance data block in the I_DB parameter. If the SFB is called as local instance, specify the number of the local instance in the OFFSET parameter (0 if there is no local instance, 1 for the first local instance, 2 for the second one, and so on).

With signal status "1" at the EN_R parameter (enable to receive), the status of the communication instance is displayed which is specified at the I_DB parameter. You must evaluate the NDR, ERROR and STATUS parameters after *every* block call.

The parameters I_TYP, I_STATE, I_CONN and I_STATUS provide information concerning the status of the local communication instance.

Table 20.24 SFB Parameters for Querying Status

Parameter	Present in SFB		Declaration	Data Type	Contents, Description
REQ	22	-	INPUT	BOOL	Start data exchange
EN_R	-	23	INPUT	BOOL	Ready to receive
ID	22	23	INPUT	WORD	Connection ID
NDR	22	23	OUTPUT	BOOL	New data fetched
ERROR	22	23	OUTPUT	BOOL	Error occurred
STATUS	22	23	OUTPUT	WORD	Job status
PHYS	22	23	IN_OUT	ANY	Physical status
LOG	22	23	IN_OUT	ANY	Logical status
LOCAL	22	23	IN_OUT	ANY	Status of an S7 CPU as partner

Table 20.25 Parameters of the Blocks FC 62 C_CNTRL and SFC 62 CONTROL

Parameter	Present in		Declaration	Data Type	Contents, Description
EN_R	FC	SFC	INPUT	BOOL	Ready to receive
I_DB	-	SFC	INPUT	BLOCK_DB	Instance data block
OFFSET	-	SFC	INPUT	WORD	Number of the local instance
ID	FC	-	INPUT	WORD	Connection ID
RET_VAL	-	SFC	RETURN	INT	Error information
RETVAL	FC	-	RETURN	INT	Error information
ERROR	FC	SFC	OUTPUT	BOOL	Error detected
STATUS	FC	SFC	OUTPUT	WORD	Status word
I_TYP	-	SFC	OUTPUT	BYTE	Block type identifier
I_STATE	-	SFC	OUTPUT	BYTE	Actual status identifier
I_CONN	-	SFC	OUTPUT	BOOL	Connection status ("1" = connection exists)
I_STATUS	-	SFC	OUTPUT	WORD	STATUS parameter for communication instance
C_CONN	FC	-	OUTPUT	BOOL	Connection status ("1" = connection exists)
C_STATUS	FC	-	OUTPUT	WORD	Connection status

FC 62 C_CNTRL

Check the status of a connection

In S7-300 systems, FC 62 C_CNTRL determines the status of a connection in the local device. Enter the connection ID in the ID parameter which STEP 7 defines in the connection table for the local device.

With signal status "1" at the EN_R parameter (enable to receive), the actual connection status is displayed. The ERROR and STATUS param-

eters must be evaluated following each block call.

The C_CONN and C_STATUS parameters provide information on the actual connection status.

SFC 87 C_DIAG

Determine the connection status

The system function SFC 87 C_DIAG determines the actual status of connections with a fixed configuration, i.e. all S7 connections and

Table 20.26 Parameters of the SFC 87 C_DIAG

Parameter	Declaration	Data Type	Contents, Description
REQ	INPUT	BOOL	Trigger a job with signal status “1”
MODE	INPUT	BYTE	Operating mode, see text
RET_VAL	RETURN	INT	Error information
BUSY	OUTPUT	BOOL	Job still in progress when “1”
N_CON	OUTPUT	INT	Index of the last structure
CON_ARR	OUTPUT	ANY	Destination range for the read connection data

all fault-tolerant S7 connections. With each call, the SFC 87 C_DIAG reads the connection data from the operating system and enters them into the user memory for evaluation. The SFC then acknowledges that reading has taken place in the operating system so that a change in status since the last read operation can be recorded. If you wish to monitor the connec-

tions permanently, call the SFC at regular intervals, e.g. every 10 seconds in a time interrupt organization block.

SFC 87 C_DIAG is a system function which operates in asynchronous mode. You trigger a job with signal status “1” and the REQ parameter. If the job cannot be executed immediately,

```
DATA_BLOCK con_data          //Data block with the connection data
...
STRUCT
...
con_req      : BOOL;        //Trigger job
con_busy     : BOOL;        //Job running
con_error    : INT;         //Error information of SFC
con_index    : INT;         //Number of field elements read by SFC
con_status   : ARRAY [1..12] OF STRUCT
    CON_ID   : WORD;        //Connection ID
    STAT_CON : BYTE;        //Connection status
    PROD_CON : BYTE;        //Partial connection number of productive connection
    STBY_CON : BYTE;        //Partial connection number of standby connection
    DIS_PCON : BOOL;        //Change in status in fault tolerance
    DIS_CON  : BOOL;        //Change in status of conn. (without fault tolerance)
    RES0     : BYTE;        //Reserved
    RES1     : BYTE;        //Reserved
END_STRUCT;
...
END_STRUCT
END_DATA_BLOCK
```

You can use your own expressions for the names of variables and components. The call could then be as follows, for example:

```
CALL C_DIAG (
    REQ      := con_data.con_req,
    MODE     := B#16#02,
    RET_VAL  := con_data.con_error,
    BUSY     := con_data.con_busy,
    N_CON    := con_data.con_index,
    CON_ARR  := con_data.con_status);
```

Figure 20.33 Programming Example of the SFC 87 C_DIAG

the SFC returns the signal status “0” on the BUSY parameter, otherwise BUSY = “1” means that the job is in progress.

Table 20.26 shows the parameters of SFC 87 C_DIAG.

SFC 87 C_DIAG can work in various operating modes which you can set using the MODE parameter:

- ▷ MODE = B#16#00
The SFC acknowledges reading out, without copying the connection data.
- ▷ MODE = B#16#01
The SFC copies the connection data and acknowledges reading out.
- ▷ MODE = B#16#02
The SFC only copies the connection data if they have changed, and acknowledges reading out – even if no changes have occurred.
- ▷ MODE = B#16#03
The SFC copies the connection data without acknowledging.

The SFC 87 C_DIAG transmits the actual connection data from the operating system into the destination area specified in the CON_ARR parameter. The destination area is a field of structures; a field component contains the data for one connection. The number of field elements (structures) must correspond to the number of possible connections. Figure 20.33 shows the possible structure of a corresponding field variable with the connection data.

The field with the connection data is not arranged according to the connection IDs; the individual connections can be assigned in any manner to the field elements. Field elements with invalid connections can also be positioned between field elements with valid connections. The data of a connection are consistent with one another.

20.8 IE communication

20.8.1 Basics

With “Open communication via Industrial Ethernet” (IE communication for short), you transfer data between two devices connected to

the Ethernet subnet. Communication can be implemented using the protocols TCP native in accordance with RFC 793, ISO-on-TCP in accordance with RFC 1006, or UDP in accordance with RFC 768.

The communication functions are loadable function blocks (FBs) contained in the *Standard Library* of STEP 7 under *Communication Blocks*. Included are user-defined data types (UDTs) with the structure of the connection data and the address of the communication partner.

Configuring IE communication

The following are required before data can be transferred with IE communication:

- ▷ In the case of the protocols TCP native and ISO-on-TCP, a connection must be established to the communication partner (“connection-oriented protocols”) or
- ▷ In the case of the protocol UDP, a connection must be established to the communication layer of the CPU operating system (“connectionless protocol”). The partner is then addressed when the relevant function block is called.

The connection is configured via a data area (not using the connection table). The necessary data structures are stored in the user-defined data type UDT 65 TCON_PAR that the function blocks use for establishing and clearing down the connection. The data contain the connection ID that states a specific connection and the associated function block calls, and the information on the protocol used.

Establishment of the connection to the partner or setting up of the communication access point is handled by the function block FB 65 TCON that you call in the main program of both partner devices. Data can be transferred in parallel in both directions over an established connection. Several connections can exist on one physical line. The function block FB 66 TDIS_CON clears the connection down again and thus releases the resources used (Figure 20.34).

With the function blocks FB 63 TSEND and FB 64 TRCV, you can transfer data with the protocols TCP native or ISO-on-TCP. Data transfer with the UDP protocol requires the function

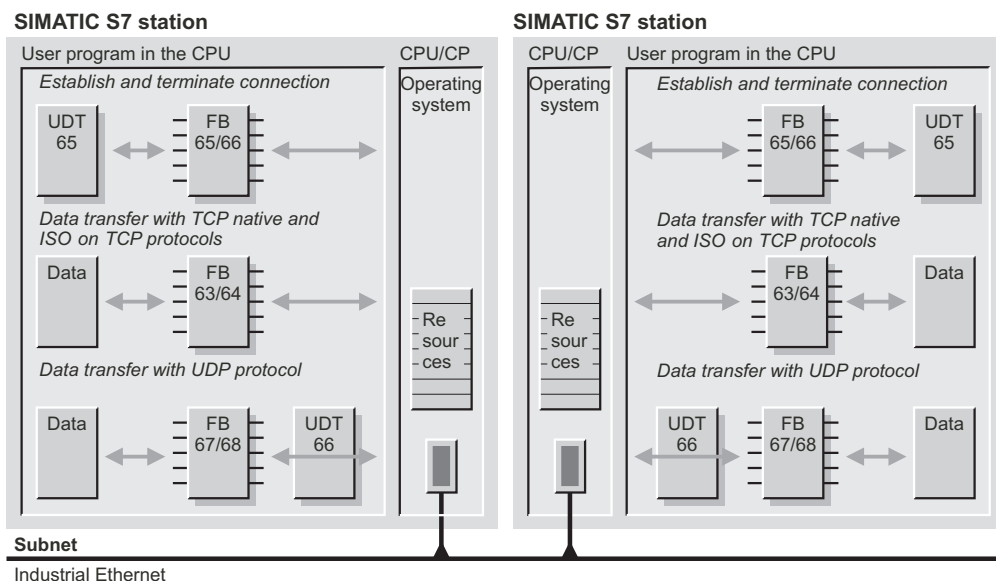


Figure 20.34 IE communication

blocks FB 67 TUSEND and FB 68 TURCV. When calling these function blocks, specify the address of the partner device in a data area. The structure of this address is in the user-defined data type UDT 66 TADD_PAR.

Calling function blocks

The function blocks for IE communication work asynchronously, that is, job execution can take several program cycles under certain circumstances. You can call the communication blocks in the main program and control data transfer with the parameters REQ and EN_R. You must evaluate the results at the parameters BUSY, NDR, DONE, ERROR and STATUS immediately after each execution since they only remain valid until the next call.

20.8.2 Establishing and clearing down connections

Before data can be transferred with IE communication, a connection must be established with the partner device (in the case of TCP native and ISO-on-TCP) or with the communication layer of the operating system (in the case of UDP). The following blocks are available for this purpose:

- ▷ FB 65 TCON
Establish connection to the communication partner or the communication layer of the operating system
- ▷ FB 66 TDISCON
Clear down connection
- ▷ UDT 65 TCON_PAR
Structure for the connection data

You can find the parameters of the function blocks in Table 20.27. The design of the data structure is shown in Table 20.28.

FB 65 TCON Establish connection

Function block FB 65 TCON creates the prerequisites for IE communication. The parameters required for this are located in a data area that has the structure of the user-defined data type UDT 65 TCON_PAR.

When using the protocols TCP native and ISO on TCP, a connection is established to the communication partner. The station for which “Active connection setup” is entered establishes the connection. The partner station must then be designated as “passive”. This designation is independent of the direction of transfer of the data.

Table 20.27 Parameters for FB 65 TCON and FB 66 TDISCON

Parameters	for FB		Declaration	Data Type	Contents, Description
REQ	65	66	INPUT	BOOL	Start job (with rising edge)
ID	65	66	INPUT	WORD	Reference to communication connection
DONE	65	66	OUTPUT	BOOL	Job running (“0”) or executed without errors (“1”)
BUSY	65	66	OUTPUT	BOOL	Job being processed (“1”) or has been completed (“0”)
ERROR	65	66	OUTPUT	BOOL	Error occurred (at “1”)
STATUS	65	66	OUTPUT	WORD	Job status, error information in the case of ERROR = “1”
CONNECT	65	-	IN_OUT	ANY	Pointer to the connection description

Table 20.28 Structure of the connection description UDT 65 TCON_PAR

Byte	Parameter	Data Type	Default value	Contents, Description
0 to 1	block-length	WORD	W#16#0040	Length of UDT 65 (64 bytes)
2 to 3	id	WORD	W#16#0000	Connection reference Value range: W#16#0001 to W16#0FFF
4	connection_type	BYTE	B#16#01	Connection type B#16#01: TCP/IP native (compatibility mode) B#16#11: TCP/IP native B#16#12: ISO on TCP B#16#13: UDP
5	active_est	BOOL	“0”	Type of connection buildup: “1” active; “0” passive; with UDP: always “0”
6	local_device_id	BYTE	B#16#02	Device identifier: ID of the communication device (see manual)
7	local_tsap_id_len	BYTE	B#16#02	Length of parameter local_tsap_id
8	rem_sub_net_id	BYTE	B#16#00	Currently not used
9	rem_staddr_len	BYTE	B#16#00	Length of the address of the remote connection point; not used with UDP
10	rem_tsap_id_len	BYTE	B#16#00	Length of parameter rem_tsap_id; not used with UDP
11	next_staddr_len	BYTE	B#16#00	Length of parameter next_staddr; not used with UDP
12 to 27	local_tsap_id	ARRAY [1..16] OF BYTE	16(B#16#00)	Local port number or local TSAP
28 to 33	rem_subnet_id	ARRAY [1..6] OF BYTE	6(B#16#00)	Currently not used
34 to 39	rem_staddr	ARRAY [1..6] OF BYTE	6(B#16#00)	IP address of the remote connection point; not used with UDP
40 to 55	rem_tsap_id	ARRAY [1..16] OF BYTE	16(B#16#00)	Remote port number or remote TSAP; not used with UDP
56 to 61	next_staddr	ARRAY [1..6] OF BYTE	6(B#16#00)	Rack and slot of the local CP; not used with UDP
62 to 63	spare	WORD	W#16#0000	Must be assigned W#16#0000

The connection is monitored and maintained by the operating system of the CPU. In the event of a break in connection, the active partner attempts to re-establish the connection without having to call FB 65 TCON again. FB 66 TDISCON clears down the communication connection in STOP mode of the CPU or in the case of POWER OFF/ON.

If the UDP protocol is used, FB 65 TCON sets up a local communication access point that represents the connection between the user program and the communication layer of the operating system. No connection is made to the connection partner.

You designate the communication connection by assigning the ID parameter. The specification must correspond to the variable *id* in the connection data. You specify the connection data with the pointer at the CONNECT parameter.

In the initial state, the parameters REQ, BUSY, DONE and ERROR have signal state “0”. You start connection setup with a rising edge on the REQ parameter.

While the job is running, BUSY = “1”. The job has been successfully completed when BUSY = “0”, DONE = “1” and ERROR = “0”. If the job contains errors, BUSY = “0”, DONE = “0” and ERROR = “1”. The error is then specified at the STATUS parameter. BUSY, DONE and ERROR are reset to “0” when REQ is returned to “0”.

FB 66 TDISCON **Establish connection**

Function block FB 66 TDISCON terminates the prerequisites for IE communication. It clears down the connection to the communication partner or deletes the communication access point.

You designate the communication connection by assigning the ID parameter. The specification must correspond to the variable *id* in the connection data.

In the initial state, the parameters REQ, BUSY, DONE and ERROR have signal state “0”. You start connection clear-down with a rising edge on the REQ parameter.

While the job is running, BUSY = “1”. The job has been successfully completed when BUSY = “0”, DONE = “1” and ERROR = “0”. If the job contains errors, BUSY = “0”, DONE = “0” and ERROR = “1”. The error is then specified at the STATUS parameter. BUSY, DONE and ERROR are reset to “0” when REQ is returned to “0”.

UDT 65 TCON_PAR **Structure of the connection data**

The user-defined data type UDT 65 TCON_PAR contains the structure of the connection data either for the communication connection to the partner device (protocols TCP native and ISO-on-TCP) or for the connection to the communication layer of the local operating system (UDP protocol).

You require a data block with this structure for each connection. For each connection, you can use your own global data block based on the UDT, or you can combine the data blocks in a shared global data block.

The assignment of the variables depends on the protocol and the devices used (see Online Help of STEP 7). UDTs with different defaults are stored in the library:

- ▷ UDT 651: for TCP active
- ▷ UDT 652: for TCP passive
- ▷ UDT 653: for ISO-on-TCP active
- ▷ UDT 654: for ISO-on-TCP passive
- ▷ UDT 655: for ISO-on-TCP active with CP
- ▷ UDT 656: for ISO-on-TCP passive with CP
- ▷ UDT 657: for open UDP local

20.8.3 Data transfer with TCP native or ISO-on-TCP

The following function blocks are available to you for data transfer with the connection-oriented protocols TCP native and ISO-on-TCP:

- ▷ FB 63 TSEND
Send data with logic connection
- ▷ FB 64 TRCV
Receive data with logic connection

You can find the parameters of these function blocks in Table 20.29.

Table 20.29 Parameters for FB 63 TSEND and FB 64 TRCV

Parameter	with FB		Declaration	Data type	Contents, Description
REQ	63	-	INPUT	BOOL	Start data transfer (with rising edge)
EN_R	-	64	INPUT	BOOL	FB ready to receive (with “1”)
ID	63	64	INPUT	WORD	Reference to communication connection
LEN	63	64	INPUT	INT	Number of bytes to be sent or received
DONE	63	-	OUTPUT	BOOL	Job running (“0”) or executed without errors (“1”)
NDR	-	64	OUTPUT	BOOL	Job running (“0”) or completed (“1”)
BUSY	63	64	OUTPUT	BOOL	Job being processed (“1”) or has been completed (“0”)
ERROR	63	64	OUTPUT	BOOL	Error occurred (at “1”)
STATUS	63	64	OUTPUT	WORD	Job status, error information in the case of ERROR = “1”
RCVD_LEN	-	64	OUTPUT	INT	Number of bytes actually received
DATA	63	64	IN_OUT	ANY	Send or receive mailbox

You have to establish the connection to the partner station with FB 65 TCON before transferring the data (see Chapter 20.8.2, “Establishing and clearing down connections”). Data can be exchanged simultaneously in both directions over the connection with FB 63 TSEND and FB 64 TRCV.

FB 63 TSEND

Send data with logic connection

The function block FB 63 TSEND sends data with the protocols TCP native or ISO on TCP via an existing communication connection.

You can designate the communication connection by assigning the ID parameter. The specification must agree with the variable *id* in the connection data. You can specify the send mailbox with the pointer at the DATA parameter.

In the initial state, the parameters REQ, BUSY, DONE and ERROR have signal state “0”. Start the data transfer with a rising edge on the REQ parameter. On the initial call with “1”, the data is fetched from the area specified with the DATA parameter. The number of bytes speci-

fied at the LEN parameter is sent. Their maximum number depends on the connection type:

While the job is running, BUSY = “1”. The job has been successfully completed when BUSY = “0”, DONE = “1” and ERROR = “0”. If the job contains errors, BUSY = “0”, DONE = “0” and ERROR = “1”. The error is then specified at the STATUS parameter. BUSY, DONE and ERROR are reset to “0” when REQ is returned to “0”.

The data in the send area can then be modified again when either DONE or ERROR has signal state “1”.

FB 64 TRCV

Receive data with logic connection

The function block FB 64 TRCV receives data with the protocols TCP native or ISO on TCP via an existing communication connection.

Designate the communication connection by assigning the ID parameter. The specification must agree with the variable *id* in the connection data. Specify the receive mailbox with the pointer at the DATA parameter.

If the *LEN* parameter has 0, the length specified in the DATA parameter is used. After a data block has been received, the number of bytes received is made available on the RCVD_LEN parameter, and NDR is set to signal state “1”.

With the *protocol TCP native*, neither the length of the message frame nor its start or end are transferred. So that the number of bytes sent is

Connection type	Number of bytes
B#16#01	1 to 1460
B#16#11	1 to 8192
B#16#12	1 to 1452 (with CP) 1 to 8192 (without CP)

correctly received, the LEN parameter at the receive mailbox must be assigned the same value as the LEN parameter at the send mailbox.

If a larger value has been selected for the LEN at the receive mailbox, part of the following message frame (from the next job) will also be received. NDR is only set to “1” when the parameterized length has been reached.

If a smaller value has been selected for LEN, NDR is set to “1” when the parameterized length is reached, and the RCVD_LEN parameter is assigned the number of received bytes. With each subsequent call, another data block is received.

Information about the length and end of a message frame is sent with the *protocol ISO-on-TCP*. If LEN at the receive mailbox is larger than at the send mailbox, the sent data are received, NDR is set to “1”, and the number of received bytes is written in RCVD_LEN. If LEN is smaller, an error message is issued: ERROR = “1”, STATUS = W#16#8088.

FB 64 TRCV only receives data when the EN_R parameter has signal state “1”.

While the job is running, BUSY = “1”. The job has been successfully completed when BUSY = “0”, NDR = “1” and ERROR = “0”. If the job contains errors, BUSY = “0”, NDR = “0” and

ERROR = “1”. The error is then specified at the STATUS parameter. BUSY, NDR and ERROR are reset to “0” when EN_R is returned to “0”.

The data in the receive mailbox is consistent when NDR has signal state “1”.

20.8.4 Data transfer with UDP

The following blocks are available to you for data transfer with the connectionless UDP protocol:

- ▷ FB 67 TUSEND
Send data with UDP
- ▷ FB 68 TURCV
Receive data with UDP
- ▷ UDT 66 TADD_PAR
Data structure for the partner address

Table 20.30 shows the parameters of the function blocks, and Table 20.31 shows the structure of the UDT.

You have to establish a connection to the communication layer of the operating system with FB 65 TCON before transferring the data (see Chapter 20.8.2, “Establishing and clearing down connections”). The address of the communication partner is located in a data area that has the structure of UDT 66 TADD_PAR.

Table 20.30 Parameters for FB 67 TUSEND and FB 68 TURCV

Parameters	with FB		Declaration	Data type	Contents, Description
REQ	67	-	INPUT	BOOL	Start data transfer (with rising edge)
EN_R	-	68	INPUT	BOOL	FB ready to receive (with “1”)
ID	67	68	INPUT	WORD	Reference to the communication point
LEN	67	68	INPUT	INT	Number of bytes to be sent or received
DONE	67	-	OUTPUT	BOOL	Job running (“0”) or executed without errors (“1”)
NDR	-	68	OUTPUT	BOOL	Job running (“0”) or completed (“1”)
BUSY	67	68	OUTPUT	BOOL	Job being processed (“1”) or has been completed (“0”)
ERROR	67	68	OUTPUT	BOOL	Error occurred (at “1”)
STATUS	67	68	OUTPUT	WORD	Job status, error information in the case of ERROR = “1”
RCVD_LEN	-	68	OUTPUT	INT	Number of bytes actually received
DATA	67	68	IN_OUT	ANY	Send or receive mailbox
ADDR	67	68	IN_OUT	ANY	Pointer to the address of the sender or receiver

Table 20.31 Structure of the partner address UDT 66 TADD_PAR

Byte	Parameter	Data type	Default value	Contents, Description
0 to 3	rem_ip_addr	ARRAY [1..4] OF BYTE	4(B#16#00)	IP address of the partner
4 to 5	rem_port_nr	ARRAY [1..2] OF BYTE	2(B#16#00)	Port number of the partner
6 to 7	spare	ARRAY [1..2] OF BYTE	2(B#16#00)	Must be assigned 0000

FB 67 TUSEND Send data with UDP

The function block FB 67 TUSEND sends data with the UDP protocol.

The assignment of the ID parameter designates the connection between the user program and the communication layer of the operating system. The value must agree with the variable *id* in the connection data. Specify the send mailbox with the pointer at the DATA parameter.

The information on the communication partner is located in a data area to which the pointer at the ADDR parameter points. With each new send job, the address and thus the partner can be changed without having to redefine the communication access point with FB 65 TCON.

In the initial state, the parameters REQ, BUSY, DONE and ERROR have signal state “0”. Start data transfer with a rising edge on the REQ parameter. On the initial call with “1”, the data is fetched from the area specified with the DATA parameter. The number of bytes specified at the LEN parameter is sent (1 to max. 1,460).

While the job is running, BUSY = “1”. The job has been successfully completed when BUSY = “0”, DONE = “1” and ERROR = “0”. If the job contains errors, BUSY = “0”, DONE = “0” and ERROR = “1”. The error is then specified at the STATUS parameter. BUSY, DONE and ERROR are reset to “0” when REQ is returned to “0”.

The data in the send area can then be modified again when either DONE or ERROR has signal state “1”.

FB 68 TURCV Receive data with UDP

The function block FB 68 TURCV receives data with the UDP protocol.

The assignment of the ID parameter designates the connection between the user program and the communication layer of the operating system. The value must agree with the variable *id* in the connection data. Specify the receive mailbox with the pointer at the DATA parameter.

The information on the communication partner is located in a data area to which the pointer at the ADDR parameter points.

The number of bytes to be received is set at the LEN parameter (1 to max. 1,460). After a data block has been received, the number of bytes received is made available on the RCVD_LEN parameter, and NDR is set to signal state “1”.

Data is only received when the EN_R parameter has signal state “1”.

While the job is running, BUSY = “1”. The job has been successfully completed when BUSY = “0”, NDR = “1” and ERROR = “0”. If the job contains errors, BUSY = “0”, NDR = “0” and ERROR = “1”. The error is then specified at the STATUS parameter. BUSY, NDR and ERROR are reset to “0” when EN_R is returned to “0”.

The data in the receive area is consistent when NDR has signal state “1”.

UDP (User Data Protocol)

Connection buildup is not carried out with UDP. The communication partner is specified at the ADDR parameter of the send block (IP address and port number). The receive block then supplies the IP address and the port number of the sender at the ADDR parameter.

The user-defined data type UDT 66 TADD_PAR contains the structure of the address information. The pointer at ADDR points to a data area with this structure.

With UDP, information on the length and end of a message frame is transferred. If LEN at the receive mailbox is larger, the sent data is copied

to the receive mailbox, NDR is set to "1", and the number of received bytes is written in RCVD_LEN. If LEN is smaller, an error message is issued: ERROR = "1", STATUS = W#16#8088.

UDT 66 TADD_PAR

Data structure of the partner address

UDT 66 contains the structure of the partner address in the case of transfer with the UDP protocol. The ADDR parameter at the function blocks FB 67 TUSEND and FB 68 TURCV points to a data area with this structure.

20.9 PtP communication with S7-300C

20.9.1 Fundamentals

When using point-to-point communication (PtP), you transmit data over a serial interface to a communications partner, e.g. a printer or a SIMATIC S5 station. With certain S7-300 compact CPUs, an RS 422/485 interface (X.27) is already integrated.

The communications connections are specified using the Hardware Configuration tool when

parameterizing the CPU's interface properties. ASCII mode, the 3964(R) procedure and the RK512 computer coupling are available as transmission protocols.

The communications functions are system function blocks SFB which are integrated in the operating system of the S7-300C CPU. The instance data blocks for these SFBs are present in the user memory. The SFBs do not test the parameters. If parameters are set incorrectly, it may occur that the CPU goes to STOP. If you use the transmission protocol of the computer coupling, a synchronization data block is additionally used (once for all computer coupling SFBs in the user memory, Figure 20.35).

Configuring PtP communication

Use the Hardware Configuration tool to set the transmission protocol in the properties window of the point-to-point interface:

▷ ASCII mode

The data are transmitted as ASCII characters. The transmission is not acknowledged. Setting of signal assignments and transmission parameters such as baud rate, parity, end-of-text character.

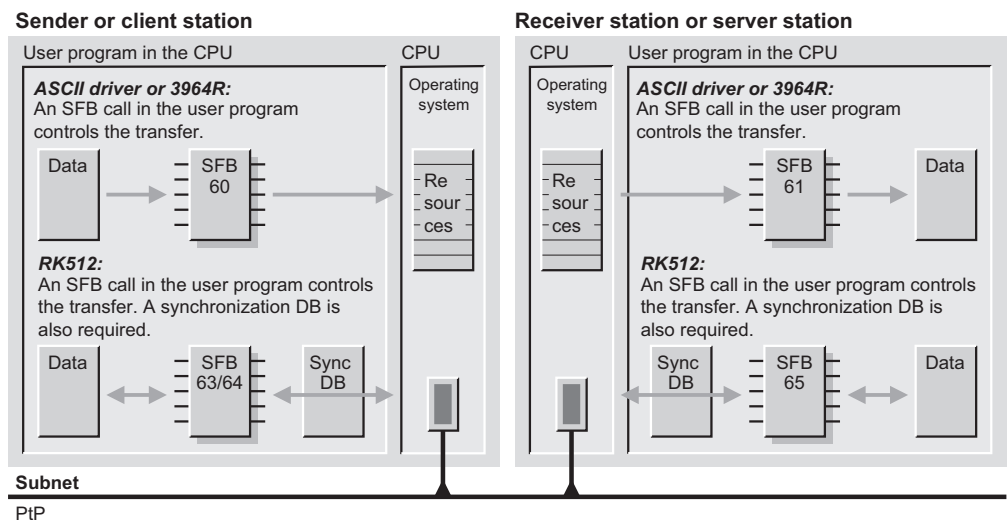


Figure 20.35 Point-to-point Connection with S7-300C

- ▷ 3964(R) procedure
The data are sent to the communications partner, and positively acknowledged by this when received correctly. Setting of signal assignments and transmission parameters such as baud rate, parity, block check.
- ▷ RK512 computer coupling
Data transmission can be coordinated by so-called communication flags. Data receipt and the fetching of data are acknowledged. Setting of signal assignments and transmission parameters such as baud rate, parity, block check.

With the 3964(R) and RK512 transmission protocols, the communications partners must have different priorities in order to define the sequence when requests to transmit are sent simultaneously.

20.9.2 ASCII driver and 3964(R) procedure

You can send and receive data via the point-to-point connection when using the ASCII driver or the 3964(R) procedure. The following system function blocks are required:

- ▷ SFB 60 SEND_PTP
Send data with ASCII driver or 3964(R) procedure

- ▷ SFB 61 RCV_PTP
Receive data with ASCII driver or 3964(R) procedure
- ▷ SFB 62 RES_RCVB
Delete receive buffer with ASCII driver or 3964(R) procedure

Table 20.32 shows the parameters of these system function blocks.

SFB 60 SEND_PTP Send data with ASCII driver or 3964R procedure

The system function block SFB 60 SEND_PTP is used to send a data area to a communications partner. Use the Hardware Configuration tool to set the transmission protocol and the transmission parameters. Specify the area of data to be transmitted using the SD_1 parameter. The length of the transmitted data area depends on the interface parameters, e.g. send up to length specified by the LEN parameter, or send up to an end-of-text character.

With the ASCII driver, you can send telegrams up to a length of 1024 bytes. SFB 60 SEND_PTP transmits the data in consistent blocks of 206 bytes. You must not change the data in the send area while the transmission is running.

Sending is triggered by a rising signal edge on the REQ parameter. With signals status “1” on

Table 20.32 SFB Parameters for Sending and Receiving Data with ASCII Driver or 3964(R) Procedure

Parameter	Present in SFB			Declaration	Data Type	Contents, Description
REQ	60	-	62	INPUT	BOOL	Trigger job with signals status “1”
EN_R	-	61	-	INPUT	BOOL	Receive enable
R	60	61	62	INPUT	BOOL	With “1”, the job is aborted
LADDR	60	61	62	INPUT	WORD	Submodule address of interface
DONE	60	-	62	OUTPUT	BOOL	With “1”, the job is still running
NDR	-	61	-	OUTPUT	BOOL	With “1”, the job has been completed without fault
ERROR	60	61	62	OUTPUT	BOOL	With “1”, a fault has occurred
STATUS	60	61	62	OUTPUT	WORD	Error information
SD_1	60	-	-	IN_OUT	ANY	Send mailbox
RD_1	-	61	-	IN_OUT	ANY	Receive mailbox
LEN	60	61	-	IN_OUT	INT	Number of transmitted bytes

the DONE parameter, the SFB signals that the job has been completed successfully. In the event of an error, the ERROR parameter is set to "1" and the error information output in the STATUS parameter.

You can abort a current send job by means of signal status "1" on the R parameter, and reset the call instance to the basic state.

SFB 61 RCV_PTP

Receive data with ASCII driver or 3964R procedure

The system function block SFB 61 RCV_PTP is used to receive a data area from a communications partner. Use the Hardware Configuration tool to set the transmission protocol and the transmission parameters. The received data are entered into the area specified by the RD_1 parameter. The number of received bytes is present in the LEN parameter.

SFB 61 RCV_PTP receives the data in consistent blocks of 206 bytes. You must not access the data in the receive area while the transmission is running.

The CPU's internal receive buffer has a size of 2048 bytes. In the parameter settings of the interface, you can also define whether you wish to use the complete length of the receive buffer for data receipt or to limit the number of received telegrams.

A signal status "1" on the EN_R parameter enables data receipt. The NDR parameter has a signal status "1" if new data have been received successfully. In the event of an error, the ERROR parameter is set to "1" and the error information output in the STATUS parameter.

You can abort a current send job by means of signal status "1" on the R parameter, and reset the call instance to the basic state.

SFB 62 RES_RCVB

Delete receive buffer with ASCII driver or 3964R procedure

System function block SFB 62 RES_RCVB deletes the receive buffer of the point-to-point interface. A telegram received during the delete operation is exempted from this.

Deleting is triggered by the rising signal edge on the REQ parameter. With signal status "1" on the DONE parameter, the SFB signals that deleting was successful. In the event of an error, the ERROR parameter is set to "1" and the error information output in the STATUS parameter. DONE, ERROR and STATUS are only set for one call at a time.

You can abort a current send job by means of signal status "1" on the R parameter, and reset the call instance to the basic state.

20.9.3 RK512 computer coupling

With the point-to-point connection, you can send and receive data with application of the RK512 computer coupling. The system function blocks required for this are:

- ▷ SFB 63 SEND_RK
Send data with RK512 computer coupling
- ▷ SFB 64 FETCH_RK
Fetch data with RK512 computer coupling
- ▷ SFB 65 SERVE_RK
Receive data and make available with RK512 computer coupling

Table 20.33 shows the parameters of these system function blocks.

Specification of transmission area

SFB 63 SEND_RK sends from the area specified by SD_1, and addresses a data block area in the partner device. SFB 64 FETCH_RK can access all address areas in the partner device, and stores the fetched data in the data block area specified by the RD_1 parameter. SFB 65 SERVE_RK can save received data in a data block, and make data available from all address areas. The permissible assignments for the parameters are listed in Table 20.34. Certain values are only meaningful in conjunction with a SIMATIC S5 station as the partner device.

Synchronization data block

The system function blocks of the computer coupling work together with a synchronization data block in addition to the instance data block, and this synchronizes and controls the activities of all computer coupling instances. The synchroniza-

Table 20.33 SFB Parameters for Sending and Receiving Data with RK512 Computer Coupling

Parameter	Present in SFB			Declaration	Data Type	Contents, Description
SYNC_DB	63	64	65	INPUT	INT	Number of the synchronization data block
REQ	63	64	-	INPUT	BOOL	Trigger job with “1”
EN_R	-	-	65	INPUT	BOOL	Enable receipt with “1”
R	63	64	65	INPUT	BOOL	Abort job with “1”
LADDR	63	64	65	INPUT	WORD	Logical basic address of interface
R_CPU	63	64	-	INPUT	INT	CPU number of partner station
R_TYPE	63	64	-	INPUT	CHAR	Type of data block in partner CPU
R_DBNO	63	64	-	INPUT	INT	Number of data block in partner CPU
R_OFFSET	63	64	-	INPUT	INT	Number of start byte in partner CPU
R_CF_BYT	63	64	-	INPUT	INT	Number of communication flag byte in partner CPU
R_CF_BIT	63	64	-	INPUT	INT	Number of communication flag bit in partner CPU
DONE	63	64	-	OUTPUT	BOOL	With “1”: job completed without error
NDR	-	-	65	OUTPUT	BOOL	With “1”: job completed without error
ERROR	63	64	65	OUTPUT	BOOL	With “1”: job completed with error
STATUS	63	64	65	OUTPUT	WORD	Error information
L_TYPE	-	-	65	OUTPUT	CHAR	Type of data area on local CPU
L_DBNO	-	-	65	OUTPUT	INT	Number of data block in local CPU
L_OFFSET	-	-	65	OUTPUT	INT	Number of start byte in local CPU
L_CF_BYT	-	-	65	OUTPUT	INT	Number of communication flag byte in local CPU
L_CF_BIT	-	-	65	OUTPUT	INT	Number of communication flag bit in local CPU
SD_1	63	-	-	IN_OUT	ANY	Send mailbox
RD_1	-	64	-	IN_OUT	ANY	Receive mailbox
LEN	63	64	65	IN_OUT	INT	Number of data bytes

tion data block is present once in the user memory. You should create it as a global data block with a minimum length of 240 bytes. Enter the number of the data block in the SYNC_DB parameter.

Coordination with communication flags

Receipt of data using the computer coupling can be coordinated by communication flags. A communication flag is a bit from the flag address area F. Use a communication flag for each transmission job, and specify its address in the R_CF_BYT and R_CF_BIT or L_CF_BYT and L_CF_BIT parameters.

If the local CPU is the client, the system function block SFB 63 SEND_RK is used to send data and SFB 64 FETCH_RK to fetch data. When sending and fetching data from the partner CPU, the address of the communication flag is also included. If this communication flag in the partner CPU has the signal status “0”, the partner CPU permits importing of the data package into the user memory in the case of a send job, and reading of the data package from the user memory in the case of a fetch job. The communication flag is then set by the communications function to indicate that the data transmission has taken place. The data can then be edited or preprocessed by the user program. If the communication flag is reset by the user

Table 20.34 Specification of Transmission Area

Parameter	Type:	SFB 63 Send data	SFB 64 Fetch data	SFB 65 Receive data	SFB 65 Provision of data	Meaning
R_CPU	INT	0 to 4	0 to 4	-	-	0 = single-processor mode 1..4 = number of CPU in multi-processor mode
R_TYPE	CHAR	D, X	D, X, M, E, A, T, C	-	-	D = data block DB X = extended data block DX M = flag memory area
L_TYPE	CHAR	-	-	D	D, M, E, A, T, Z	E = process image of inputs A = process image of outputs T = timer values Z = counter values
R_DBNO	INT	0 to 255	0 to 255	-	-	Number of data block (irrelevant with M, E, A, T and Z)
L_DBNO	INT	-	-	1 to n ¹⁾	1 to n ¹⁾	
R_OFFSET	INT	0 to 510	0 to 510	-	-	First byte with data blocks (must be an even address)
		-	0 to 255	-	-	First byte with M, E, A, T and Z
L_OFFSET	INT	-	-	0 to 1024	0 to 1024	Telegram length

¹⁾ CPU-specific

program, the data transmission is enabled again. The communication flag in the partner CPU therefore permits control of data transmission.

If the local CPU is the server, the data are received by SFB 65 SERVE_RK if the client sends data, or made available if the client fetches data. On the SFB, parameterize the local communication flag (in the server) with which you then control the receipt or provision of data in the user program. The SFB indicates in the L_CF_BYT and L_CF_BIT parameters which communication flag is used for the currently executed job and has been set to "1". Following processing of the data (fetching or provision again), reset the communication flag by the program, thus enabling processing of the next transmission job.

SFB 63 SEND_RK

Send data with RK512 computer coupling

The system function block SFB 63 SEND_RK is used to send a data area to a communications partner. Use the Hardware Configuration tool to set the transmission protocol and the transmission parameters. Specify the data area to be sent in the SD_1 parameter. Enter the length of

the sent data area in the LEN parameter. Note that the number of bytes must be even.

SFB 63 SEND_RK only transmits the data if the communication flag in the communications partner has the signal status "0". A telegram can be up to 1024 bytes long. The data are transmitted in consistent blocks of 128 bytes. You must not change the data in the send area while a transmission is taking place.

Sending is triggered by the rising signal edge on the REQ parameter. With signal status "1" on the DONE parameter, the SFB signals that the job was completed successfully. In the event of an error, the ERROR parameter is set to "1" and the error information output in the STATUS parameter.

You can abort a current send job by means of signal status "1" on the R parameter, and reset the call instance to the basic state.

SFB 64 FETCH_RK

Fetch data with the RK512 computer coupling

The system function block SFB 64 FETCH_RK is used to fetch a data area from a communications partner. Use the Hardware Configu-

ration tool to set the transmission protocol and the transmission parameters. The fetched data are entered into the area specified by the RD_1 parameter. The number of received bytes is present in the LEN parameter.

SFB 64 FETCH_RK only fetches the data if the communication flag in the communications partner has the signal status "0". A telegram can be up to 1024 bytes long. The data are transmitted in consistent blocks of 128 bytes. You must not change the data in the send area while a transmission is taking place.

You enable the fetching of data by means of signal status "1" in the EN_R parameter. Signal status "1" in the NDR parameter indicates that new data have been fetched successfully. In the event of an error, the ERROR parameter is set to "1" and the error information output in the STATUS parameter.

You can abort a current fetch job by means of signal status "1" on the R parameter, and reset the call instance to the basic state.

SFB 65 SERVE_RK

Receive data and make available with the RK512 computer coupling

The system function block SFB 65 SERVE_RK has the server functionality for the RK512 computer coupling. It accepts a data area sent by a communications partner, and makes a data area available which is fetched by a communications partner. The received data, or data made available, are entered into the area specified by the L_TYPE, L_DBNO and L_OFFSET parameters. The number of transmitted bytes is present in the LEN parameter.

SFB 65 SERVE_RK transmits the data in consistent blocks of 128 bytes. You must not access the data in the transmission area while a transmission is taking place. Coordination of data transmission is controlled by a communication flag in the user program.

You enable processing of the job by signal status "1" in the EN_R parameter. Signal status "1" in the NDR parameter indicates that new data have been successfully received or fetched. In the event of an error, the ERROR parameter is set to "1" and the error information output in the STATUS parameter.

You can abort a current job by means of signal status "1" on the R parameter, and reset the call instance to the basic state.

20.10 Configuration in RUN

Configuration in RUN (CiR) means that changes to the system can be made during operation. This functionality permits you to change the configuration of the distributed I/O of an S7 station without the CPU entering STOP or having to be set to STOP.

The possible changes include the addition of compact DP slaves, ET200M stations and PA master systems to an existing DP master system, the addition of modules to ET200M stations, and the addition of PA slaves (field devices) to existing PA master systems. All objects added during runtime can also be removed again (Figure 20.36).

Prerequisites and limitations exist in addition to the fact that all involved devices must be able to handle the CiR functionality. For example, the PROFIBUS DP master system must be a mono-master system and must not exhibit constant bus cycle times, the use of intelligent DP slaves is not permissible in the associated station components, and the module parameters must be saved on the CPU.

Components with and without CiR functionality can be mixed; however, changes are only possible on components with CiR capability.

During reconfiguration, processing is interrupted for a short period (typically 1 s, can be programmed). The time can be kept short if only a few changes are always carried out.

20.10.1 Preparation of Changes in Configuration

Configure, for example, an S7-400 station (CPU firmware release V3.1 or higher) with at least one PROFIBUS DP master system using the hardware configuration tool. Then add the DP slaves and – if envisaged – the dummy values for later system expansion (CiR object under PROFIBUS DP in the hardware catalog). Set the later maximum configuration in the

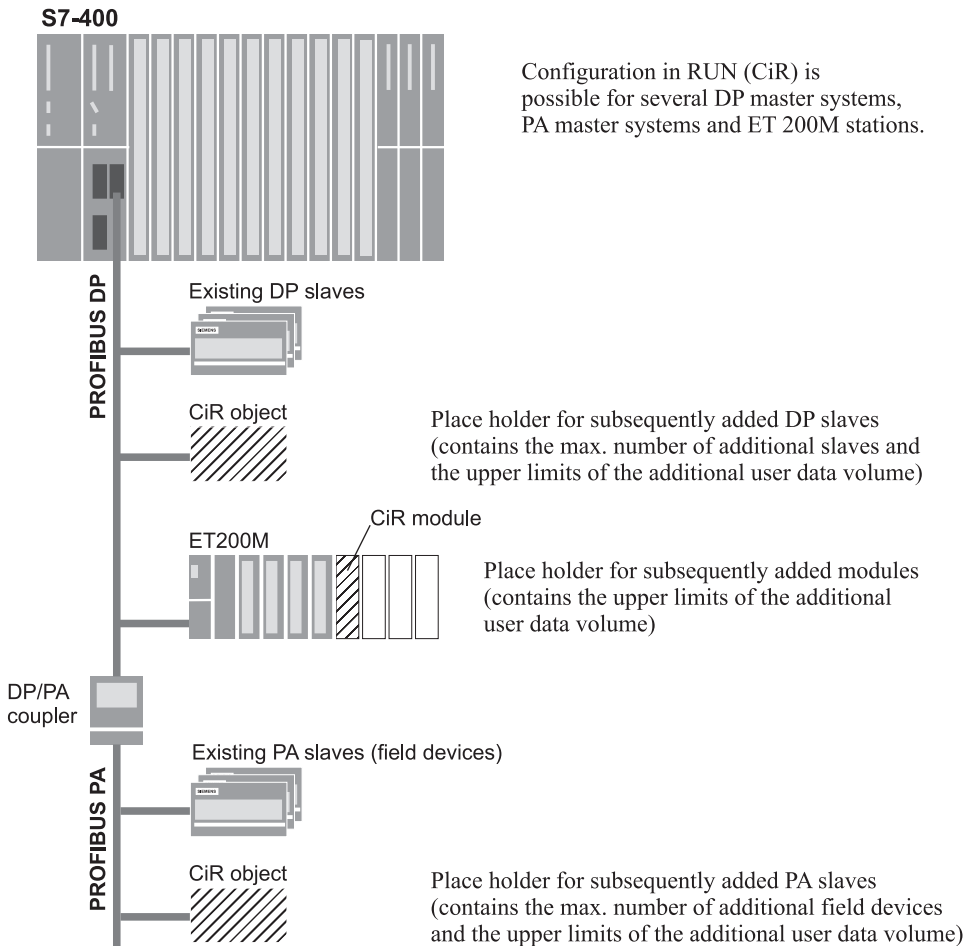


Figure 20.36 CiR Elements in the Hardware Configuration

properties of this dummy value. The Hardware Configuration tool calculates 244 input bytes and 244 output bytes for each additional DP slave. In reality, far fewer user data are usually required. You can change the total of all required input and output bytes by clicking the control box “Enhanced settings”.

You can also provide space in an ET200M station for later expansions. Add a station with an interface module IM 153-2BA00-0XB0 or better to the DP master system, and activate the checkbox “Module replacement during runtime” on the “Special” tab in the properties of the station. The active bus modules required to carry out the module replacements are dis-

played in the bottom window area (they are required for the mechanical design, but are not configured). It is recommendable to fit the ET200M station with active bus modules up to the planned configuration since they must not be inserted or removed during runtime.

You can now insert modules into the ET200M station and – if you wish to carry out later expansions – also a dummy directly following the last configured module (the object *CiR module* under the used IM 153 in the hardware catalog). Set the required number of additional input and output bytes in the properties of the CiR module.

If you wish to extend a PA master system connected to the DP master system, use an IM 157-0AA82-0XA0 or better as the DP/PA link. A dummy is also provided for this for subsequent expansion with field devices (*CiR object* under the used IM 157 in the hardware catalog). Set the required number of additional input and output bytes in the properties of the CiR module.

With the CiR-capable DP master system marked and by using EDIT → MASTER SYSTEM → ENABLE CiR CAPABILITY, a CiR object is generated on the DP master system and on every subordinate PA master system with CiR capability. A CiR module is inserted into each ET200M station with CiR capability. All CiR objects and CiR modules are deleted again using EDIT → MASTER SYSTEM → DISABLE CiR CAPABILITY.

20.10.2 Change Configuration

You can now change the configuration within the limits specified in the CiR elements, and load again in RUN. Possible modifications are:

- ▷ Adding of compact and modular DP slaves to an existing DP master system (the added slaves must have a higher PROFIBUS address than the largest address previously used)
- ▷ Changing of assignment of partial process image with existing DP slaves
- ▷ Adding of PA slaves (field devices) to an existing PA master system
- ▷ Adding of DP/PA couplers after an IM 157 interface module
- ▷ Adding of DP/PA links including PA master system to an existing DP master system
- ▷ Adding of modules to an ET200M station
- ▷ Changing of module parameters in an ET200M station (e.g. new or modified assignment to a partial process image, activation of previously unused channels)
- ▷ Cancellation of above-mentioned modifications (starting at the highest addresses for modules and slaves)

The total of configured (real, immediately used) addresses and the addresses for future use must not be greater than the quantity framework of the DP master (is checked during configuring), but can be larger than the quantity framework of the CPU (is only checked when “converting” into specific slaves or modules).

During a CiR process, the configuration can be changed on a maximum of 4 DP master systems. Under certain conditions it is recommendable or even essential to carry out the CiR process more than once. If, for example, modules or slaves are to be replaced by others, the corresponding component must be removed first, and the replacement added in a second CiR process.

20.10.3 Load Configuration

The (initial) loading of a configuration with CiR elements or with a modified CiR configuration is carried out with the CPU in the STOP status. In order to check whether the CiR capability is also present, you should subsequently load the CiR configuration again in RUN. Testing for CiR capability using STATION → CHECK CiR COMPATIBILITY is not 100% possible offline. For example, the CiR synchronization time could be limited by the SFC 104 CiR.

To keep the CPU in the RUN status during the CiR process, you must make sure that interrupts from unknown components are ignored. A corresponding program must be present in the following organization blocks:

- ▷ Process interrupts OB 40 to OB 47
- ▷ Timing error OB 80
- ▷ Diagnostic interrupt OB 82
- ▷ Hot swapping interrupt OB 83
- ▷ Program execution fault OB 85
- ▷ Rack failure OB 86
- ▷ I/O access fault OB 122

When adding modules or slaves, you should first load the configuration and then the matching user program. When removing modules and slaves, first load the adapted user program and then the modified configuration. Adding or removing real modules or slaves must only be

carried out following loading of the modified configuration (when the INTF LED on the CPU has gone off).

When reparameterizing modules you must first load a user program which no longer addresses the associated modules or no longer evaluates their interrupts. Then load the modified configuration, change the hardware if necessary, and load the user program adapted to the modification.

20.10.4 CiR Synchronization Time

Following loading of the new configuration into the CPU, the new data are checked and imported into the current configuration if correct. This importing requires a certain time, the so-called CiR synchronization time. Process execution is interrupted during this period.

The CiR synchronization time is calculated from the total of the CiR synchronization times of all DP and PA master systems involved. The synchronization time for a master system depends on the CPU used and on the real and planned I/O volumes in this master system. This time is indicated by the Hardware Configuration tool in the properties of the CiR object in the master system. The worst case is always calculated, so that the actual CiR synchronization time is shorter. The synchronization time is 100 ms if only modules are reparameterized.

The CPU compares the calculated CiR synchronization time with the permissible upper limit, whose default setting is 1000 ms. You can change this upper limit using the SFC 104 CIR. If the calculated CiR synchronization time is greater than this upper limit, the change in configuration is not carried out.

20.10.5 Effects on Program Execution

Execution of the user program is stopped during the CiR synchronization time. All process images retain their last values. The SIMATIC timers and the CPU clock continue. Any inter-

rupts which occur are only processed at the end of the CiR synchronization time. Communication with a connected programming device is limited; only the STOP command is accepted.

Following synchronization, the CPU starts the organization block OB 80 "Timing error" with the value W#16#350A in the first word of the start information (variables OB80_EV_CLASS and OB80_FLT_ID). The required CiR synchronization time in ms is present in the variable OB80_ERROR_INFO.

If modules are to be reparameterized, the CPU then starts the organization block OB 83 "Hot swapping interrupt" with the value W#16#3367 in the first word of the start information (variables OB83_EV_CLASS and OB83_FLT_ID). The modules are subsequently reparameterized. It can occur that the associated modules do not deliver valid values in the meantime.

Following reparameterization, the CPU starts the OB 83 again, this time with W#16#3267 in the first word of the start information. Faulty reparameterization is signaled by W#16#3968. The associated modules are then considered as not available. The described procedure is executed in each master system which is affected.

20.10.6 Control CiR Process

You can use the **SFC 104 CIR** to block the CiR process in the user program, to limit it for a certain period, or to enable it. The SFC parameters are listed in Table 20.35.

MODE = B#16#00 delivers the currently valid upper limit for the CiR synchronization time. MODE = B#16#01 sets the CiR synchronization time to the default value 1000 ms and enables execution of the CiR process. MODE = B#16#02 unconditionally disables the CiR process, MODE = B#16#03 only disables it if the CiR synchronization time calculated in the CPU is greater than that specified in the FRZ_TIME parameter.

Table 20.35 Parameters of the SFC 104 CIR

Parameter	Declaration	Data Type	Contents, Description
MODE	INPUT	BYTE	Job identification B#16#00: information function B#16#01: enable CiR process B#16#02: disable CiR process B#16#03: conditionally disable CiR process
FRZ_TIME	INPUT	TIME	Upper limit of CiR synchronization time Default setting: T#1000 ms Permissible range: T#200ms to T#2500ms
RET_VAL	RETURN	INT	Error information
A_FT	OUTPUT	TIME	Currently valid upper limit of CiR synchronization time

21 Interrupt Handling

Interrupt handling is always event-driven. When such an event occurs, the operating system interrupts scanning of the main program and calls the routine allocated to this particular event. When this routine has executed, the operating system resumes scanning of the main program at the point of interruption. Such an interruption can take place after every operation (statement).

Applicable events may be interrupts and errors. The order in which virtually simultaneous interrupt events are handled is regulated by a priority scheduler. Each event has a particular servicing priority. Several interrupt events can be combined into priority classes.

Every routine associated with an interrupt event is written in an organization block in which additional blocks can be called. A higher-priority event interrupts execution of the routine in an organization block with a lower priority. You can affect the interruption of a program by high-priority events using system functions.

21.1 General Remarks

SIMATIC S7 provides the following interrupt events (interrupts):

- ▷ Time-of-day interrupt
An interrupt generated by the operating system at a specific time of day, either once only or periodically
- ▷ Time-delay interrupt
An interrupt generated after a specific amount of time has passed; a system function call determines the instant at which this time period begins
- ▷ Watchdog interrupt
An interrupt generated by the operating system at periodic intervals

- ▷ Hardware interrupt
An interrupt from a module, either via an input derived from a process signal or generated on the module itself
- ▷ DPV1 interrupt
An interrupt from a PROFIBUS DPV1 slave
- ▷ Multiprocessor interrupt
An interrupt generated by another CPU in a multiprocessor network
- ▷ Synchronous cycle interrupt
An interrupt from a PROFIBUS DP master synchronous to the DP cycle

Other interrupt events are the synchronous errors which may occur in conjunction with program scanning and the asynchronous errors, such as diagnostic interrupts. The handling of these events is discussed in 23, "Error Handling".

Priorities

An event with a higher priority interrupts a program being processed with lower priority because of another event. The main program has the lowest priority (priority class 1), asynchronous errors the highest (priority class 26), apart from the start-up routine. All other events are in the intervening priority classes. In S7-300 systems, the priorities are fixed; in S7-400 systems, you can change the priorities by parameterizing the CPU accordingly.

An overview of all priority classes, together with the default organization blocks for each, is presented in Chapter 3.1.2, "Priority Classes".

Disabling interrupts

The organization blocks for event-driven program scanning can be disabled and enabled with system functions SFC 39 DIS_IRT and SFC 40 EN_IRT and delayed and enabled with

SFC 41 DIS_AIRT and SFC 42 EN_AIRT (see Chapter 21.9, “Handling Interrupt Events”).

Current signal states

In an interrupt handling routine, one of the requirements is that you work with the current signal states of the I/O modules (and not with the signal states of the inputs that were updated at the start of the main program) and write the fetched signal states direct to the I/O (not waiting until the process-image output table is updated at the end of the main program).

In the case of a few inputs and outputs for the interrupt handling routine, it is enough to access the I/O modules direct with load and transfer operations (AWL) or with the MOVE box (LAD, FBD). You are recommended here to maintain a strict separation between the main program and the interrupt handling routine with regard to the I/O signals.

If you want to process many input and output signals in the interrupt handling routine, the solution on the S7-400 CPUs is to use subprocess images. When assigning addresses, you assign each module to a subprocess image. With SFC 26 UPDAT_PI and SFC 27 UPDAT_PO, you update the subprocess images in the user program (see also Chapter 20.2.1, “Process Image Updating”).

On new S7-400 CPUs, you can assign an input and an output subprocess image to each interrupt organization block (each interrupt priority class) and so cause the process images to be updated automatically when the interrupt occurs.

Start information, temporary local data

Every organization block delivers the start information in the first 20 bytes of its temporary local data. You can generate the declaration of the start information using own data yourself, or you can use the templates from the *Standard Library* under *Organization Blocks*.

In S7-300 systems, the available temporary local data have a fixed length of 256 bytes per priority class. In S7-400 systems, you can specify the length per priority class by parameterizing the CPU accordingly (parameter block

“local data”), whereby the total may not exceed a CPU-specific maximum. Note that the minimum number of bytes for temporary local data for the priority class used must be 20 bytes so as to be able to accommodate the start information. Specify zero for unused priority classes.

Note that you can only directly read the start information of an organization block in the block itself, since temporary local data are involved. If you also require values from the start information in blocks which are located in lower call levels, call the system function SFC 6 RD_SINFO at the corresponding point in the program (see Chapter 20.2.5, “Start Information”).

Actual interrupt information

Bytes 4 to 11 of the start information in the interrupt organization block contain the information specific to the triggered interrupt. In many cases, the components triggering the interrupt deliver additional information which you can then read in the interrupt organization block using the system function block SFB 54 RALRM (see Chapter 21.9.3, “Reading additional Interrupt Information”).

21.2 Time-of-Day Interrupts

Time-of-day interrupts are used when you want to run a program at a particular time, either once only or periodically, for instance daily. In STEP 7, organization blocks OB 10 to OB 17 are provided for servicing time-of-day interrupts; which of these eight organization blocks are actually available depends on the CPU used.

You can configure the time-of-day interrupts in the hardware configuration data or control them at runtime via the program using system functions. The prerequisite for proper handling of the time-of-day interrupts is a correctly set real-time clock on the CPU.

Table 21.1 shows the start information for the time-of-day interrupts. The dummy value xx represents the number of the associated interrupt organization block 10 to 17.

Table 21.1 Start Information for Time-of-Day Interrupts

Byte	Variable Name	Data Type	Description	Contents
0	OBxx_EV_CLASS	BYTE	Event class	B#16#11 = UP event
1	OBxx_STRT_INF	BYTE	Start request for the interrupt OB	B#16#11 = OB 10
2	OBxx_PRIORITY	BYTE	Priority class	Default value 2 for all time-of-day interrupts
3	OBxx_OB_NUMBR	BYTE	OB number	B#16#xx
4	OBxx_RESERVED_1	BYTE	Reserved	-
5	OBxx_RESERVED_2	BYTE	Reserved	-
6..7	OBxx_PERIOD_EXE	WORD	Interval with OBs called periodically	See description of SFC 28 SET_TINT
8..9	OBxx_RESERVED_3	INT	Reserved	-
10..11	OBxx_RESERVED_4	INT	Reserved	-
12..19	OBxx_DATE_TIME	DATE_AND_TIME	Occurrence of event	Called time of OB

xx represents the OB numbers 10 to 17

21.2.1 Handling Time-of-Day Interrupts

General remarks

To start a time-of-day interrupt, you must first set the start time, then activate the interrupt. You can perform the two activities separately via the hardware configuration data or using SFCs. Note that when activated via the hardware configuration data, the time-of-day interrupt is started automatically following parameterization of the CPU.

You can start a time-of-day interrupt in two ways:

- ▷ Single-shot: the relevant OB is called once only at the specified time, or
- ▷ Periodically: depending on the parameter assignments, the relevant OB is started every minute, hourly, daily, weekly, monthly or yearly.

Following a single-shot time-of-day interrupt OB call, the time-of-day interrupt is canceled. You can also cancel a time-of-day interrupt with SFC 29 CAN_TINT.

If you want to once again use a canceled time-of-day interrupt, you must set the start time again, then reactivate the interrupt.

You can query the status of a time-of-day interrupt with SFC 31 QRY_TINT.

Performance characteristics during startup

During a cold restart or warm restart, the operating system clears all settings made with SFCs. Settings made via the hardware configuration data are retained. On a hot restart, the CPU resumes servicing of the time-of-day interrupts in the first complete scan cycle of the main program.

You can query the status of the time-of-day interrupts in the start-up OB by calling SFC 31, and subsequently cancel or re-set and reactivate the interrupts. The time-of-day interrupts are serviced only in RUN mode.

Performance characteristics on error

If a time-of-day interrupt OB is called but was not programmed, the operating system calls OB 85 (program execution error). If OB 85 was not programmed, the CPU goes to STOP.

Time-of-day interrupts that were deselected when the CPU was parameterized cannot be serviced, even when the relevant OB is available. The CPU goes to STOP.

If you activate a time-of-day interrupt on a single-shot basis, and if the start time has already passed (from the real-time clock's point of view), the operating system calls OB 80 (timing

error). If OB 80 is not available, the CPU goes to STOP.

If you activate a time-of-day interrupt on a periodic basis, and if the start time has already passed (from the real-time clock's point of view), the time-of-day interrupt OB is executed the next time that time period comes due.

If you set the real-time clock ahead by more than approx. 20 s, whether for the purpose of correction or synchronization, thus skipping over the start time for the time-of-day interrupt, the operating system calls OB 80 (timing error). The time-of-day interrupt OB is then executed precisely once.

If you set the real-time clock back by more than approx. 20 s, whether for the purpose of correction or synchronization, an activated time-of-day interrupt OB will no longer be executed at the instants which are already past.

If a time-of-day interrupt OB is still executing when the next (periodic) call occurs, the operating system invokes OB 80 (timing error). When OB 80 and the time-of-day interrupt OB have executed, the time-of-day interrupt OB is restarted.

Disabling, delaying and enabling

Time-of-day interrupt OB calls can be disabled and enabled with SFC 39 DIS_IRT and SFC 40 EN_IRT, and delayed and enabled with SFC 41 DIS_AIRT and SFC 42 EN_AIRT.

21.2.2 Configuring Time-of-Day Interrupts with STEP 7

The time-of-day interrupts are configured via the hardware configuration data. Open the selected CPU with EDIT → OBJECT PROPERTIES and choose the “Time-of-Day” tab from the dialog box.

In S7-300 controllers, the processing priority is permanently set to 2. In S7-400 controllers, you can set a priority between 2 and 24, depending on the CPU, for each possible OB; priority 0 deselects an OB. You should not assign a priority more than once, as interrupts might be lost

when more than 12 interrupt events with the same priority occur simultaneously.

The “Active” option activates automatic starting of the time-of-day interrupt. The “Execution” option screens a list which allows you to choose whether you want the OB to execute on a single-shot basis or at specific intervals. The final parameter is the start time (date and time).

When it saves the hardware configuration, STEP 7 writes the compiled data to the *System Data* object in the offline user program *Blocks*. From here, you can load the parameter assignment data into the CPU while the CPU is at STOP; these data then go into force immediately.

21.2.3 System Functions for Time-of-Day Interrupts

The following system functions can be used for time-of-day interrupt control:

- ▷ SFC 28 SET_TINT
Set time-of-day interrupt
- ▷ SFC 29 CAN_TINT
Cancel time-of-day interrupt
- ▷ SFC 30 ACT_TINT
Activate time-of-day interrupt
- ▷ SFC 31 QRY_TINT
Query time-of-day interrupt

The parameters for these system functions are listed in Table 21.2.

SFC 28 SET_TINT Set time-of-day interrupt

You determine the start time for a time-of-day interrupt by calling system function SFC 28 SET_TINT. SFC 28 sets only the start time; to start the time-of-day interrupt OB, you must activate the time-of-day interrupt with SFC 30 ACT_TINT. Specify the start time in the SDT parameter in the format DATE_AND_TIME, for instance DT#1997-06-30-08:30. The operating system ignores seconds and milliseconds and sets these values to zero. Setting the start time will overwrite the old start time value, if any. An active time-of-day interrupt is canceled, that is, it must be reactivated

Table 21.2 SFC Parameters for Time-of-Day Interrupts

SFC	Parameter	Declaration	Data Type	Contents, Description
28	OB_NR	INPUT	INT	Number of the OB to be called at the specified time on a single-shot basis or periodically
	SDT	INPUT	DT	Start date and start time in the format DATE_AND_TIME
	PERIOD	INPUT	WORD	Period on which start time is based: W#16#0000 = Single-shot W#16#0201 = Every minute W#16#0401 = Hourly W#16#1001 = Daily W#16#1201 = Weekly W#16#1401 = Monthly W#16#2001 = Last in the month W#16#1801 = Yearly
	RET_VAL	RETURN	INT	Error information
29	OB_NR	INPUT	INT	Number of the OB whose start time is to be deleted
	RET_VAL	RETURN	INT	Error information
30	OB_NR	INPUT	INT	Number of the OB to be activated
	RET_VAL	RETURN	INT	Error information
31	OB_NR	INPUT	INT	Number of the OB whose status is to be queried
	RET_VAL	RETURN	INT	Error information
	STATUS	OUTPUT	WORD	Status of the time-of-day interrupt

SFC 30 ACT_TINT**Activate time-of-day interrupt**

A time-of-day interrupt is activated by calling system function SFC 30 ACT_TINT. When a TOD interrupt is activated, it is assumed that a time has been set for the interrupt. If, in the case of a single-shot interrupt, the start time is already past, SFC 30 reports an error. In the case of a periodic start, the operating system calls the relevant OB at the next applicable time. Once a single-shot time-of-day interrupt has been serviced, it is, for all practical purposes, canceled. You can re-set and reactivate it (for a different start time) if desired.

SFC 29 CAN_TINT**Cancel time-of-day interrupt**

You can delete a start time, thus deactivating the time-of-day interrupt, with system function SFC 29 CAN_TINT. The respective OB is no longer called. If you want to use this same time-

of-day interrupt again, you must first set the start time, then activate the interrupt.

SFC 31 QRY_TINT**Query time-of-day interrupt**

You can query the status of a time-of-day interrupt by calling system function SFC 31 QRY_TINT. The required information is returned in the STATUS parameter.

When the bits have signal state "1", they have the following meanings:

- 0 CPU is starting up
- 1 The interrupt has been disabled by the call of SFC 39 DIS_IRT
- 2 Time-of-day interrupt is activated and has not elapsed
- 3 (always "0")
- 4 An organization block with the number of OB_NR is loaded
- 5 (always "0")

21.3 Time-Delay Interrupts

A time-delay interrupt allows you to implement a delay timer independently of the standard timers. In STEP 7, organization blocks OB 20 to OB 23 are set aside for time-delay interrupts; which of these four organization blocks are actually available depends on the CPU used.

The priorities for time-delay interrupt OBs are programmed in the hardware configuration data.

Table 21.3 shows the start information for the time-delay interrupts. The dummy value xx represents the number of the associated interrupt organization block 20 to 23.

21.3.1 Handling Time-Delay Interrupts

General remarks

A time-delay interrupt is started by calling SFC 32 SRT_DINT; this system function also passes the delay interval and the number of the selected organization block to the operating system. When the delay interval has expired, the OB is called.

You can cancel servicing of a time-delay interrupt, in which case the associated OB will no longer be called.

You can query the status of a time-delay interrupt with SFC 34 QRY_DINT.

Performance characteristics during startup

On a cold restart or warm restart, the operating system deletes all programmed settings for time-delay interrupts. On a hot restart, the settings are retained until processed in RUN mode, whereby the “residual cycle” is counted as part of the start-up routine.

You can start a time-delay interrupt in the start-up routine by calling SFC 32. When the delay interval has expired, the CPU must be in RUN mode in order to be able to execute the relevant organization block. If this is not the case, the CPU waits to call the organization block until the start-up routine has terminated, then calls the time-delay interrupt OB before the first network in the main program.

Performance characteristics on error

If no time-delay interrupt OB has been programmed, the operating system calls OB 85 (program execution error). If there is no OB 85 in the user program, the CPU goes to STOP.

Table 21.3 Start Information for Time-Delay Interrupts

Byte	Variable Name	Data Type	Description	Contents
0	OBxx_EV_CLASS	BYTE	Events class	B#16#11 = UP event
1	OBxx_STRT_INF	BYTE	Start request for the interrupt OB	B#16#21 = OB 20
2	OBxx_PRIORITY	BYTE	Priority class	Default values 3 to 6 (OB 20 to OB 23)
3	OBxx_OB_NUMBR	BYTE	OB number	B#16#xx
4	OBxx_RESERVED_1	BYTE	Reserved	-
5	OBxx_RESERVED_2	BYTE	Reserved	-
6..7	OBxx_SIGN	WORD	Job identification	See description of SFC 32 SRT_DINT
8..11	OBxx_DTIME	TIME	Expired delay time	See description of SFC 32 SRT_DINT
12..19	OBxx_DATE_TIME	DATE_AND_TIME	Occurrence of event	Call time of OB

xx represents the OB numbers 20 to 23

If the delay interval has expired and the associated OB is still executing, the operating system calls OB 80 (timing error) or goes to STOP if there is no OB 80 in the user program.

Time-delay interrupts which were deselected during CPU parameterization cannot be serviced, even when the respective OB has been programmed. The CPU goes to STOP.

Disabling, delaying and enabling

The time-delay interrupt OBs can be disabled and enabled with system functions SFC 39 DIS_IRT and SFC 40 EN_IRT, and delayed and enabled with SFC 41 DIS_AIRT and SFC 42 EN_AIRT.

21.3.2 Configuring Time-Delay Interrupts with STEP 7

Time-delay interrupts are configured in the hardware configuration data. Simply open the selected CPU with EDIT → OBJECT PROPERTIES and choose the “Interrupts” tab from the dialog box.

In S7-300 controllers, the priority is permanently preset to 3. In S7-400 controllers, you can choose a priority between 2 and 24, depending on the CPU, for each possible OB; choose priority 0 to deselect an OB. You should not assign a priority more than once, as interrupts could be lost if more than 12 interrupt events with the same priority occur simultaneously.

When it saves the hardware configuration, STEP 7 writes the compiled data to the *System Data* object in the offline user program *Blocks*. From here, you can transfer the parameter assignment data while the CPU is at STOP; the data take effect immediately.

21.3.3 System Functions for Time-Delay Interrupts

A time-delay interrupt can be controlled with the following system functions:

- ▷ SFC 32 SRT_DINT
Start time-delay interrupt
- ▷ SFC 33 CAN_DINT
Cancel time-delay interrupt
- ▷ SFC 34 QRY_DINT
Query time-delay interrupt

The parameters for these system functions are listed in Table 21.4.

SFC 32 SRT_DINT Start time-delay interrupt

A time-delay interrupt is started by calling system function SFC 32 SRT_DINT. The SFC call is also the start time for the programmed delay interval. When the delay interval has expired, the CPU calls the programmed OB and passes the time delay value and a job identifier in the start information for this OB. The job identifier is specified in the SIGN parameter for SFC 32;

Table 21.4 SFC Parameters for Time-Delay Interrupts

SFC	Parameter	Declaration	Data Type	Contents, Description
32	OB_NR	INPUT	INT	Number of the OB to be called when the delay interval has expired
	DTIME	INPUT	TIME	Delay interval; permissible: T#1ms to T#1m
	SIGN	INPUT	WORD	Job identification in the respective OB's start information when the OB is called (arbitrary characters)
	RET_VAL	RETURN	INT	Error information
33	OB_NR	INPUT	INT	Number of the OB to be canceled
	RET_VAL	RETURN	INT	Error information
34	OB_NR	INPUT	INT	Number of the OB whose status is to be queried
	RET_VAL	RETURN	INT	Error information
	STATUS	OUTPUT	WORD	Status of the time-delay interrupt

you can read the same value in bytes 6 and 7 of the start information for the associated time-delay interrupt OB. The time delay is set in increments of 1 ms. The accuracy of the time delay is also 1 ms. Note that execution of the time-delay interrupt OB may itself be delayed when organization blocks with higher priorities are being processed when the time-delay interrupt OB is called. You can overwrite a time delay with a new value by recalling SFC 32. The new time delay goes into force with the SFC call.

SFC 33 CAN_DINT Cancel time-delay interrupt

You can call system function SFC 33 CAN_DINT to cancel a time-delay interrupt, in which case the programmed organization block is not called.

SFC 34 QRY_DINT Query time-delay interrupt

System function SFC 34 QRY_DINT informs you about the status of a time-delay interrupt. You select the time-delay interrupt via the OB number, and the status information is returned in the STATUS parameter.

When the bits have signal state “1”, they have the following meanings:

- 0 CPU is starting up

- 1 The interrupt has been disabled by the call of SFC 39 DIS_IRT
- 2 The time-delay interrupt is activated and has not elapsed
- 3 (always "0")
- 4 An organization block with the number of OB_NR is loaded
- 5 (always "0")

21.4 Watchdog Interrupts

A watchdog interrupt is an interrupt which is generated at periodic intervals and which initiates execution of a watchdog interrupt OB. A watchdog interrupt allows you to execute a particular program periodically, independently of the processing time of the cyclic program.

In STEP 7, organization blocks OB 30 to OB 38 have been set aside for watchdog interrupts; which of these nine organization blocks are actually available depends on the CPU used.

Watchdog interrupt handling is set in the hardware configuration data when the CPU is parameterized.

Table 21.5 shows the start information for the watchdog interrupts. The dummy value xx represents the number of the associated interrupt organization block 30 to 38.

Table 21.5 Start Information for Watchdog Interrupts

Byte	Variable Name	Data Type	Description	Contents
0	OBxx_EV_CLASS	BYTE	Event class	B#16#11 = UP event
1	OBxx_STRT_INF	BYTE	Start request for the interrupt OB	B#16#31 = OB 30
2	OBxx_PRIORITY	BYTE	Priority class	Default values 7 to 15 (OB 30 to 38)
3	OBxx_OB_NUMBR	BYTE	OB number	B#16#xx
4	OBxx_RESERVED_1	BYTE	Reserved	-
5	OBxx_RESERVED_2	BYTE	Reserved	-
6..7	OBxx_PHS_OFFSET	INT	Phase offset	ms, see Table 21.6
8..9	OBxx_RESERVED_3	INT	Reserved	-
10..11	OBxx_EXC_FREQ	INT	Time interval	ms, see Table 21.6
12..19	OBxx_DATE_TIME	DATE_AND_TIME	Occurrence of event	Call time of OB

xx represents the OB numbers 30 to 38

21.4.1 Handling Watchdog Interrupts

Triggering watchdog interrupts in an S7-300

In an S7-300, there is a limited choice of watchdog interrupts with fixed priority, depending on the CPU. You can set the interval in the range from 1 millisecond to 1 minute, in 1-millisecond increments, by parameterizing the CPU accordingly (an interval of 500 μ s on the CPU 319 from firmware version V2.6).

Triggering watchdog interrupts in an S7-400

You define a watchdog interrupt when you parameterize the CPU. A watchdog interrupt has three parameters: the interval, the phase offset, and the priority. You can set all three. Specifiable values for interval and phase offset are from 1 millisecond to 1 minute, in 1-millisecond increments; the priority may be set to a value between 2 and 24 or to zero, depending on the CPU (zero means the watchdog interrupt is not active).

STEP 7 provides the organization blocks listed in Table 21.6, in their maximum configurations.

Phase offset

You can use the phase offset to process cyclic interrupt programs in a precise time frame even if they have the same time interval or a common multiple thereof. Use of the phase offset achieves a higher interval accuracy.

The start time of the time interval and the phase offset is the instant of transition from START-

Table 21.6 Defaults for Watchdog Interrupts

OB	Time Interval	Phase	Priority
30	5 s	0 ms	7
31	2 s	0 ms	8
32	1 s	0 ms	9
33	500 ms	0 ms	10
34	200 ms	0 ms	11
35	100 ms	0 ms	12
36	50 ms	0 ms	13
37	20 ms	0 ms	14
38	10 ms	0 ms	15

UP to RUN. The call instant for a watchdog interrupt OB is thus the time interval plus the phase offset. Figure 21.1 shows an example of this. In the left section, no phase offset is set, and consequently start of processing of the lower priority organization block is delayed by the current processing time of the higher priority organization block in each case.

If, on the other hand, a phase shift is configured and it is greater than the maximum processing time of the higher-priority organization block, the lower-priority organization block is processed in the precise time frame.

Performance characteristics during startup

Watchdog interrupts cannot be serviced in the start-up OB. The time intervals do not begin until a transition is made to RUN mode.

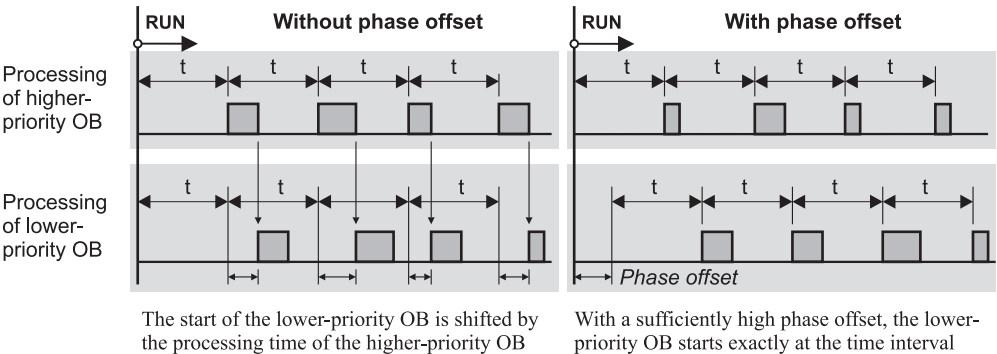


Figure 21.1 Example of Phase Offset for Watchdog Interrupts

Performance characteristics on error

When the same watchdog interrupt is generated again while the associated watchdog interrupt handling OB is still executing, the operating system calls OB 80 (timing error). If OB 80 has not been programmed, the CPU goes to STOP.

The operating system saves the watchdog interrupt that was not serviced, servicing it at the next opportunity. Only one unserviced watchdog interrupt is saved per priority class, regardless of how many unserviced watchdog interrupts accumulate.

Watchdog interrupts that were deselected when the CPU was parameterized cannot be serviced, even when the corresponding OB is available. The CPU goes to STOP in this case.

Disabling, delaying and enabling

Calling of the watchdog interrupt OBs can be disabled and enabled with system functions SFC 39 DIS_IRT and SFC 40 EN_IRT and delayed and enabled with SFC 41 DIS_AIRT and SFC 42 EN_AIRT.

21.4.2 Configuring Watchdog Interrupts with STEP 7

Watchdog interrupts are configured via the hardware configuration data. Simply open the selected CPU with EDIT → OBJECT PROPERTIES and choose the “Cyclic Interrupt” tab from the dialog box.

In S7-300 controllers, the processing priority is permanently set to 12. In S7-400 controllers, you may set a priority between 2 and 24 for each possible OB (CPU-specific); priority 0 deselects the OB to which it is assigned. You should not assign a priority more than once, as interrupts might be lost if more than 12 interrupt events with the same priority occur simultaneously.

The interval for each OB is selected under “Execution”, the delayed call instant under “Phase Offset”.

When it saves the hardware configuration, STEP 7 writes the compiled data to the *System Data* object in the offline user program *Blocks*.

From here, you can load the parameter assignment data into the CPU while the CPU is at STOP; the data take effect immediately.

21.5 Hardware Interrupts

Hardware interrupts are used to enable the immediate detection in the user program of events in the controlled process, making it possible to respond with an appropriate interrupt handling routine. STEP 7 provides organization blocks OB 40 to OB 47 for servicing hardware interrupts; which of these eight organization blocks are actually available, however, depends on the CPU.

Hardware interrupt handling is programmed in the hardware configuration data. With system functions SFC 55 WR_PARM, SFC 56 WR_DPARM and SFC 57 PARM_MOD, you can (re)parameterize the modules with hardware interrupt capability even in RUN mode.

Table 21.7 shows the start information for the process interrupts. The dummy value xx represents the number of the associated interrupt organization block 40 to 47.

21.5.1 Generating a Hardware Interrupt

A hardware interrupt is generated on the modules with this capability. This could, for example, be a digital input module that detects a signal from the process or a function module that generates a hardware interrupt because of an activity taking place on the module.

By default, hardware interrupts are disabled. A parameter is used to enable servicing of a hardware interrupt (static parameter), and you can specify whether the hardware interrupt should be generated for a coming event, a leaving event, or both (dynamic parameter). Dynamic parameters are parameters which you can modify at runtime using SFCs.

In an intelligent DP slave equipped for this purpose, you can initiate a process interrupt in the master CPU with SFC 7 DP_PRAL.

The hardware interrupt is acknowledged on the module when the organization block containing the service routine for that interrupt has finished executing.

Table 21.7 Start Information for Hardware Interrupts

Byte	Variable Name	Data Type	Description	Contents
0	OBxx_EV_CLASS	BYTE	Event class	B#16#11 = UP event
1	OBxx_STRT_INF	BYTE	Start request for the interrupt OB	B#16#41: OB 40
2	OBxx_PRIORITY	BYTE	Priority class	Default values 16 to 23 (OB 40 to 47)
3	OBxx_OB_NUMBR	BYTE	OB number	B#16#xx
4	OBxx_RESERVED_1	BYTE	Reserved	-
5	OBxx_IO_FLAG	BYTE	I/O identification	B#16#54 = input module, input submodule B#16#55 = output module, output submodule
6..7	OBxx_MDL_ADDR	WORD	Module starting address of components triggering interrupt	
8..11	OBxx_POINT_ADDR	DWORD	Interrupt information	
12..19	OBxx_DATE_TIME	DATE AND_TIME	Occurrence of event	Call time of OB

xx represents the OB numbers 40 to 47

Resolution on the S7-300

If an event occurs during execution of a hardware interrupt OB which itself would trigger generation of the same hardware interrupt, that hardware interrupt will be lost when the event that triggered it is no longer present following acknowledgment. It makes no difference whether the event comes from the module whose hardware interrupt is currently being serviced or from another module.

A diagnostic interrupt can be generated while a hardware interrupt is being serviced. If another hardware interrupt occurs on the same channel between the time the first hardware interrupt was generated and the time that interrupt was acknowledged, the loss of the latter interrupt is reported via a diagnostic interrupt to system diagnostics.

Resolution on the S7-400

If during execution of a hardware interrupt OB an event occurs on the same channel on the same module which would trigger the same hardware interrupt, that interrupt is lost. If the event occurs on another channel on the same

module or on another module, the operating system restarts the OB as soon as it has finished executing.

21.5.2 Servicing Hardware Interrupts

Querying interrupt information

The start address of the module that triggered the hardware interrupt is in bytes 6 and 7 of the hardware interrupt OB's start information. If this address is an input address, byte 5 of the start information contains B#16#54; otherwise it contains B#16#55. If the module in question is a digital input module, bytes 8 to 11 contain the status of the inputs; for any other type of module, these bytes contain the interrupt status of the module.

Interrupt handling in the start-up routine

In the start-up routine, the modules do not generate hardware interrupts. Interrupt handling begins with the transition to RUN mode. Any hardware interrupts pending at the time of the transition are lost.

Error handling

If a hardware interrupt is generated for which there is no hardware interrupt OB in the user program, the operating system calls OB 85 (program execution error). The hardware interrupt is acknowledged. If OB 85 has not been programmed, the CPU goes to STOP.

Hardware interrupts deselected when the CPU was parameterized cannot be serviced, even when the OBs for these interrupts have been programmed. The CPU goes to STOP.

Disabling, delaying and enabling

Calling of the hardware interrupt OBs can be disabled and enabled with system functions SFC 39 DIS_IRT and SFC 40 EN_IRT, and delayed and enabled with SFC 41 DIS_AIRT and SFC 42 EN_AIRT.

21.5.3 Configuring Hardware Interrupts with STEP 7

Hardware interrupts are programmed in the hardware configuration data. Open the selected CPU with EDIT → OBJECT PROPERTIES and choose the “Interrupts” tab in the dialog box.

In S7-300 systems, the default priority for OB 40 is fixed. In S7-400 systems, you can choose a priority between 2 and 24 for every possible OB (on a CPU-specific basis); priority 0 deselects execution of an OB. You should never assign the same priority twice because interrupts can be lost when more than 12 interrupt events with the same priority occur simultaneously.

You must also enable the triggering of hardware interrupts on the respective modules. To this purpose, these modules are parameterized much the same as the CPU.

When it saves the hardware configuration, STEP 7 writes the compiled data to the *System Data* object in offline user program *Blocks*; from here, you can load the parameterization data into the CPU while the CPU is in STOP mode. The parameterization data for the CPU go into force immediately following loading; the parameter assignment data for the modules take effect after the next start-up.

21.6 DPV1 Interrupts

PROFIBUS DPV1 slaves can trigger the following interrupts in addition to the types previously known with SIMATIC S7:

- ▷ Status interrupt, e.g. if the DPV1 slave changes its operating mode; the interrupt organization block OB 55 is called.
- ▷ Update interrupt, e.g. if the DPV1 slave has been parameterized over the PROFIBUS or directly; the interrupt organization block OB 56 is called.
- ▷ Vendor interrupt if an event envisaged by the vendor occurs in the DPV1 slave; the interrupt organization block OB 57 is called. The events which can trigger the interrupt are defined by the vendor of the DPV1 slave.

The origin of the interrupt, the interrupt specifier and the length of interrupt information additionally available are specified in the start information of the DPV1 interrupt organization blocks (Table 21.8). You can read the supplementary interrupt information using SFB 54 RALRM (see Chapter 21.9.3, “Reading additional Interrupt Information”).

Performance characteristics during startup

PROFIBUS DPV1 slaves can also generate interrupts when the master CPU is at STOP. In this state, the master CPU cannot call an interrupt organization block; processing of the missed interrupts is not carried out when the CPU enters the RUN state.

However, the received interrupt events are entered into the diagnostics buffer and the module status data. You can read the module status data using the system function SFC 51 RDSYSST.

Error handling

If the corresponding DPV1 interrupt OB is missing in the user program when the DPV1 interrupt is triggered, the operating system calls the OB 85 (program execution fault). The DPV1 interrupt is acknowledged. If the OB 85 is not present, the CPU enters the STOP state.

Table 21.8 Start Information for DPV1 Interrupts

Byte	Variable Name	Data Type	Description	Contents
0	OBxx_EV_CLASS	BYTE	Event class	B#16#11 = UP event
1	OBxx_STRT_INF	BYTE	Start request for the OB xx	B#16#xx
2	OBxx_PRIORITY	BYTE	Priority class	B#16#02 = default value
3	OBxx_OB_NUMBR	BYTE	OB number	B#16#xx
4	OBxx_RESERVED_1	BYTE	Reserved	-
5	OBxx_IO_FLAG	BYTE	I/O identification	B#16#54 = input module, input submodule B#16#55 = output module, output submodule
6..7	OBxx_MDL_ADDR	WORD	Module starting address of component triggering the interrupt	
8	OBxx_LEN	BYTE	Length of interrupt data record	
9	OBxx_TYPE	BYTE	Identification of interrupt type	B#16#00 = reserved B#16#01 = diagnostic interrupt B#16#02 = hardware interrupt B#16#03 = removal interrupt B#16#04 = insertion interrupt B#16#05 = status interrupt B#16#06 = update interrupt B#16#07..1F = reserved B#16#20..7E = vendor interrupt B#16#7F = reserved
10	OBxx_SLOT	BYTE	Slot number of component triggering the interrupt	
11	OBxx_SPEC	BYTE	Specifier	Bits 1 and 0: 0 0 reserved 0 1 UP event 1 0 DOWN event with error 1 1 DOWN event with further errors Bit 2: 0 no additional acknowledgment required 1 additional acknowledgment required Bits 3 to 7: reserved
12.. 19	OBxx_DATE_TIME	DATE AND TIME	Occurrence of event	Call time of OB

xx represents the OB numbers 55, 56 or 57

Disabling, delaying and enabling

Calling the DPV1 interrupt OBs can be disabled and enabled using SFC 39 DIS_IRT and SFC 40 EN_IRT respectively, and delayed and enabled using SFC 41 DIS_AIRT and SFC 42 EN_AIRT respectively.

Configuring DPV1 interrupts with STEP 7

The DPV1 interrupts are configured using the Hardware Configuration tool. Open the selected CPU using EDIT → OBJECT PROPERTIES, and select the “Interrupts” tab in the properties window which is then displayed.

The default priority is 2. You can set the priority between 2 and 24. Priority 0 deselects the interrupt. DPV1 interrupts which have been deselected cannot be executed, even if the corresponding OB is present. The CPU then enters the STOP status.

You must additionally parameterize interrupt triggering on the corresponding DPV1 slaves.

When saving the hardware configuration, STEP 7 writes the compiled data into the *System data* object in the offline user program *Blocks*; from here, you can download the parameterization data to the CPU when in the STOP status. The parameterization data for the

CPU are immediately effective following downloading, those for the DPV1 slaves following the next startup.

21.7 Multiprocessor Interrupt

The multiprocessor interrupt allows a synchronous response to an event in all CPUs in multicomputing. A multiprocessor interrupt is triggered using SFC 35 MP_ALM. Organization block OB 60, which has a fixed priority of 25, is the OB used to service a multiprocessor interrupt.

Table 21.9 shows the assignments of the start information for the multi-processor interrupt.

General remarks

An SFC 35 MP_ALM call initiates execution of the multiprocessor interrupt OB. If the CPU is in single-processor mode, OB 60 is started immediately. In multicomputing, OB 60 is started simultaneously on all participating CPUs, that is to say, even the CPU in which SFC 35 was called waits before calling OB 60 until all the other CPUs have indicated that they are ready.

Table 21.9 Start Information for the Multi-processor Interrupt

Byte	Variable Name	Data Type	Description	Contents
0	OB60_EV_CLASS	BYTE	Event class	B#16#11 = UP event
1	OB60_STRT_INF	BYTE	Start request for the OB 60	B#16#61: multi-processor interrupt triggered by own CPU B#16#62: multi-processor interrupt triggered by another CPU
2	OB60_PRIORITY	BYTE	Priority class	B#16#19 = default value (25dec)
3	OB60_OB_NUMBR	BYTE	OB number	B#16#3C (60dec)
4	OB60_RESERVED_1	BYTE	Reserved	-
5	OB60_RESERVED_2	BYTE	Reserved	-
6..7	OB60_JOB	INT	Job identification	Input variable JOB of SFC 35 MP_ALM
8..9	OB60_RESERVED_3	INT	Reserved	-
10..11	OB60_RESERVED_4	INT	Reserved	-
12..19	OB60_DATE_TIME	DATE_AND_TIME	Occurrence of event	Call time of OB

Table 21.10 Parameters for SFC 35 MP_ALM

Parameter	Declaration	Data Type	Contents, Description
JOB	INPUT	BYTE	Job identification in the range B#16#00 to B#16#0F
RET_VAL	RETURN	INT	Error information

The multiprocessor interrupt is not programmed in the hardware configuration data; it is already present in every CPU with multicomputing capability. Despite this fact, however, a sufficient number of local data bytes (at least 20) must still be reserved in the CPU's "Local Data" tab under priority class 25.

Performance characteristics during startup

The multiprocessor interrupt is triggered only in RUN mode. An SFC 35 call in the start-up routine terminates after returning error 32 929 (W#16#80A1) as function value.

Performance characteristics on error

If OB 60 is still in progress when SFC 35 is recalled, the system function returns error code 32 928 (W#16#80A0) as function value. OB 60 is not started in any of the CPUs.

The unavailability of OB 60 in one of the CPUs at the time it is called or the disabling or delaying of its execution by system functions has no effect, nor does SFC 35 report an error.

Disabling, delaying and enabling

The multiprocessor OB can be disabled and enabled with system functions SFC 39 DIS_IRT and SFC 40 EN_IRT respectively, and delayed and enabled with SFC 41 DIS_AIRT and SFC 42 EN_AIRT respectively.

SFC 35 MP_ALM

Multiprocessor interrupt

A multiprocessor interrupt is triggered with system function SFC 35 MP_ALM. Its parameters are listed in Table 21.10.

The JOB parameter allows you to forward a job identifier. The same value can be read in bytes 6 and 7 of OB 60's start information in all CPUs.

21.8 Synchronous Cycle Interrupts

Reference is made to isochrone mode if a program is executed in synchronism with the PROFIBUS DP cycle. In conjunction with constant bus cycle times, this results in reproducible response times of equal length to the I/O. The user program executed in isochrone mode is present in the organization blocks OB 61 to OB 64. The system functions SFC 126 SYNC_PI and SFC 127 SYNC_PO are available for updating of the process image in isochrone mode.

You configure isochrone mode using the Hardware Configuration tool (see section, "Configuring constant bus cycle times and isochrone mode" in Chapter 20.4.3, "Special Functions for PROFIBUS DP"). Table 21.11 shows you the start information for the synchronous cycle interrupts. The dummy value xx represents the number of the associated interrupt organization block 61 to 64.

21.8.1 Processing the Synchronous Cycle Interrupts

The synchronous cycle interrupt is triggered by the Global Control command of the DP master.

Synchronous cycle interrupts are only processed in the operating mode RUN. An asynchronous cycle interrupt in the STARTUP, STOP or HALT states is rejected. The number of OB calls which have not yet been executed is present in the start information of the isochrone mode OB when called for the first time in RUN mode.

Error handling

If an synchronous cycle interrupt arrives before the associated synchronous cycle interrupt OB has been completed, a timing error is signaled. This can occur if the user program dwells too long in an synchronous cycle interrupt OB or if

Table 21.11 Start Information for Synchronous Cycle Interrupts

Byte	Variable Name	Data Type	Description	Contents
0	OBxx_EV_CLASS	BYTE	Event class	B#16#11 = UP event
1	OBxx_STRT_INF	BYTE	Start request for the OB xx	B#16#36
2	OBxx_PRIORITY	BYTE	Priority class	B#16#19 = default value (25dec)
3	OBxx_OB_NUMBR	BYTE	OB number	B#16#xx
4	OBxx_RESERVED_1	BYTE	Reserved	-
5	OBxx_RESERVED_2	BYTE	Reserved	-
6.0	OBxx_GC_VIOL	BOOL	GC violation	See text
6.1	OBxx_FIRST	BOOL	First execution following STARTUP or HALT	With “1”
7	OBxx_MISSED_EXEC	BYTE	Number of rejected OB calls	See text
8	OBxx_DP_ID	BYTE	DP master system ID of isochronous DP master system	Is configured using the Hardware Configuration tool
9	OBxx_RESERVED_3	BYTE	Reserved	-
10..11	OBxx_RESERVED_4	WORD	Reserved	-
12..19	OBxx_DATE_TIME	DT	Occurrence of event	Call time of OB

execution has been interrupted for too long as a result of program components of higher priority. The OB called by the “too early” interrupt is rejected, and the OB 80 “Timing error” is called. A reaction can then be made here to the timing error. The number of failed synchronous cycle interrupts is output in the next processed synchronous cycle interrupt OB in the start info.

In the event of an error, the DP master can omit the Global Control command (GC) or send it offset. This “GC violation” is displayed in the start information of the next synchronous cycle interrupt OB which is called correctly.

Disabling, delaying and enabling

Calling of the synchronous cycle interrupt OBs can be disabled and enabled with the system functions SFC 39 DIS_IRT and SFC 40 EN_IRT respectively, and delayed and enabled with SFC 41 DIS_AIRT and SFC 42 EN_AIRT respectively.

21.8.2 Isochrone Updating Of Process Image

The Hardware Configuration tool can be used to assign partial process images to an synchronous cycle interrupt OB. These images are not updated automatically. The system functions **SFC 126 SYNC_PI** and **SFC 127 SYNC_PO** must be used to update the inputs and outputs respectively. Updating is carried out in isochrone mode and with data consistency. The two SFCs must only be called in an synchronous cycle interrupt OB. Direct access to the peripheral inputs and outputs of these process image partitions should be avoided.

Table 21.12 shows the parameters of the SFC 126 SYNC_PI and SFC 127 SYNC_PO.

The partial process images are not updated if an error is detected. Exceptions:

- ▷ If an access error occurs when updating the partial process image of the inputs, the inputs of faulty modules are set to “0”; the OB 85 “Program execution error” is not called.

Table 21.12 Parameters of the SFCs for Isochrone Updating Of Process Image

With SFC		Parameter Name	Declaration	Data Type	Contents, Description
126	127	PART	INPUT	BYTE	Number of partial process image B#16#01 to B#16#1E
126	127	RET_VAL	RETURN	INT	Error information
126	127	FLADDR	OUTPUT	WORD	In the event of an access error, the address of the first byte causing the error

- ▷ A consistency warning is generated if it was not possible to transmit the complete data consistently to the outputs. However, the data of individual slaves are consistent.
- ▷ If an access error occurs when updating the partial process image of the outputs, the data of the faulty modules are not transmitted; they remain unchanged in the partial process image. Updating of the unaffected modules is distributed between two DP cycles (consistency warning).

When saving the hardware configuration, STEP 7 writes the compiled data into the object *System data* in the offline user program *Blocks*; You can download the parameterization data to the CPU from here in the STOP status. The parameterization data for the CPU are immediately effective following downloading, those for the DP components following the next startup.

21.8.3 Configuration of Synchronous Cycle Interrupts with STEP 7

Configuration of synchronous cycle interrupts is carried out using the Hardware Configuration tool. Open the selected CPU with EDIT → OBJECT PROPERTIES and select the “Synchronous cycle interrupts” tab in the properties window which is then displayed.

The default priority is 25. You can set the priority between 2 and 26. Priority 0 deselects the interrupt. Synchronous cycle interrupts which have been deselected cannot be executed, even if the corresponding OB is present. In addition, you assign the isochronous DP master system and the involved partial process images to the interrupt OB.

In addition, you assign the isochronous DP master system or the isochronous PROFINET IO system and the involved process image partitions to the interrupt OB (see sections, “Configuring constant bus cycle times and isochrone mode” in Chapter 20.4.3, “Special Functions for PROFIBUS DP” or “Isochronous mode” in Chapter 20.4.6, “Special Functions for PROFINET IO”).

21.9 Handling Interrupt Events

The system functions for disabling, delaying and enabling affect all interrupts and all asynchronous errors. System functions SFC 36 to SFC 38 are provided for handling synchronous errors.

21.9.1 Disabling and Enabling interrupts

The following system functions are available for disabling and enabling interrupts and asynchronous errors:

- ▷ SFC 39 DIS_IRT
Disable interrupts
- ▷ SFC 40 EN_IRT
Enable disabled interrupts

Table 21.13 lists the parameters for these system functions.

SFC 39 DIS_IRT Disabling interrupts

System function SFC 39 DIS_IRT disables servicing of new interrupts and asynchronous errors. All new interrupts and asynchronous errors are rejected. If an interrupt or asynchronous error occurs following a Disable, the orga-

Table 21.13 SFC Parameters for Interrupt Handling

SFC	Parameter	Declaration	Data Type	Contents, Description
39	MODE	INPUT	BYTE	Disable mode (see text)
	OB_NR	INPUT	INT	OB number (see text)
	RET_VAL	RETURN	INT	Error information
40	MODE	INPUT	BYTE	Enable mode (see text)
	OB_NR	INPUT	INT	OB number (see text)
	RET_VAL	RETURN	INT	Error information
41	RET_VAL	RETURN	INT	(New) number of delays
42	RET_VAL	RETURN	INT	Number of delays remaining

nization block is not executed; if the OB does not exist, the CPU does not go to STOP.

The Disable remains in force for all priority classes until it is revoked with SFC 40 EN_IRT. After a cold or warm restart, all interrupts and asynchronous errors are enabled.

The MODE and OB_NR parameters are used to specify which interrupts and asynchronous errors are to be disabled. MODE = B#16#00 disables all interrupts and asynchronous errors. MODE = B#16#01 disables an interrupt class whose first OB number is specified in the OB_NR parameter.

For example, MODE = B#16#01 and OB_NR = 40 disables all hardware interrupts; OB = 80 would disable all asynchronous errors. MODE = B#16#02 disables the interrupt or asynchronous error whose OB number you entered in the OB_NR parameter.

Regardless of a Disable, the operating system enters each new interrupt or asynchronous error in the diagnostic buffer.

SFC 40 EN_IRT Enabling disabled interrupts

System function SFC 40 EN_IRT enables the interrupts and asynchronous errors disabled with SFC 39 DIS_IRT. An interrupt or asynchronous error occurring after the Enable will be serviced by the associated organization block; if that organization block is not in the user program, the CPU goes to STOP (except in the case of OB 81 “Power supply errors”).

The MODE and OB_NR parameters specify which interrupts and asynchronous errors are to

be enabled. MODE = B#16#00 enables all interrupts and asynchronous errors. MODE = B#16#01 enables an interrupt class whose first OB number is specified in the OB_NR parameter. MODE = B#16#02 enables the interrupt or asynchronous error whose OB number you entered in the OB_NR parameter.

21.9.2 Delaying and Enabling Interrupts

The following system functions are available for delaying and enabling interrupts and asynchronous errors:

- ▷ SFC 41 DIS_AIRT
Delay interrupts
- ▷ SFC 42 EN_AIRT
Enable delayed interrupts

Table 21.13 lists the parameters for these system functions.

SFC 41 DIS_AIRT Delaying Interrupts

System function SFC 41 DIS_AIRT delays the servicing of higher-priority new interrupts and asynchronous errors. Delay means that the operating system saves the interrupts and asynchronous errors which occurred during the delay and services them when the delay interval has expired. Once SFC 41 has been called, the program in the current organization block (in the current priority class) will not be interrupted by a higher-priority interrupt; no interrupts or asynchronous errors are lost.

Table 21.14 Parameters of System Function Block SFB 54 RALRM

Parameter	Declaration	Data Type	Contents, Description
MODE	INPUT	INT	Operating mode: 0 = shows the component triggering the interrupt 1 = writes all output parameters 2 = checks whether the selected component has triggered the interrupt
F_ID	INPUT	DWORD	Module starting address of component to be scanned
MLEN	INPUT	INT	Maximum number of bytes of supplementary interrupt information to be scanned
NEW	INPUT	BOOL	TRUE = a new interrupt has been received
STATUS	OUTPUT	DWORD	Error identification
ID	OUTPUT	DWORD	Module starting address of component triggering the interrupt
LEN	OUTPUT	INT	Number of bytes of received supplementary interrupt information
TINFO	IN_OUT	ANY	Target area for OB start information and administration information
AINFO	IN_OUT	ANY	Target area for header information and supplementary interrupt information

A delay remains in force until the current OB has terminated its execution or until SFC 42 EN_AIRT is called.

You can call SFC 41 several times in succession. The RET_VAL parameter shows the number of calls. You must call SFC 42 precisely the same number of times as SFC 41 in order to reenable the interrupts and asynchronous errors.

SFC 42 EN_AIRT Enabling delayed interrupts

System function SFC 42 EN_AIRT reenables the interrupts and asynchronous errors delayed with SFC 41. You must call SFC 42 precisely the same number of times as you called SFC 41 (in the current OB). The RET_VAL parameter shows the number of delays still in force; if RET_VAL is = 0, the interrupts and asynchronous errors have been reenabled.

If you call SFC 42 without having first called SFC 41, RET_VAL contains the value 32896 (W#16#8080).

21.9.3 Reading additional Interrupt Information

The system function block **SFB 54 RALRM** reads additional interrupt information – if pres-

ent – from the components (modules or sub-modules) triggering the interrupt. The SFB is called in an interrupt organization block or in a block called within this. Processing of the SFB 54 RALRM is carried out synchronously, i.e. the requested data are present in the output parameters immediately following the call. Table 21.14 lists the block parameters of the SFB 54 RALRM.

The SFB 54 RALRM can basically be called in all organization blocks or execution levels for all events. If you call it in an organization block whose start event is not an interrupt from the I/O, correspondingly less information is available. Depending on the respective organization block and the components triggering the interrupt, different information is entered into the target areas specified by the TINFO and AINFO parameters (Table 21.15).

The target area TINFO (task information) contains the complete start information in bytes 0 to 19 of the organization block in which the SFB 54 RALRM was called, independent of the nesting depth in which it was called. Therefore the SFB 54 RALRM partially replaces the system function SFC 6 RD_SINFO. Administration information (e.g. which component triggered the interrupt) is present in bytes 20 to 27.

The target area AINFO (interrupt information) contains the header information (e.g. the number of received bytes of supplementary inter-

Table 21.15 Assignment of TINFO and AINFO Parameters

Interrupt Type	OB No.	TINFO		AIFO	
		OB start information Bytes 0 to 19	Administration information Bytes 20 to 27	Header information Bytes 0 to 3 ¹⁾	Supplementary interrupt information Bytes 4 to 223 ¹⁾
Central hardware interrupt	40 to 47	yes	yes	yes	no
Distributed hardware interrupt	40 to 47	yes	yes	yes	As delivered by the station
Status interrupt	55	yes	yes	yes	yes
Update interrupt	56	yes	yes	yes	yes
Vendor interrupt	57	yes	yes	yes	yes
I/O redundancy error	70	yes	yes	no	no
Central diagnostic interrupt	82	yes	yes	yes	Diagnostics data record 1
Distributed diagnostic interrupt	82	yes	yes	yes	As delivered by the station
Central hot swapping interrupt	83	yes	yes	yes	no
Distributed hot swapping interrupt	83	yes	yes	yes	As delivered by the station
Rack/station failure	86	yes	yes	no	no
All other events		yes	no	no	no

With PROFINET IO:

¹⁾ 0 to 25

²⁾ 26 to 1431

rupt information or the type of interrupt) in bytes 0 to 3 and the component-specific additional interrupt information itself in bytes 4 to 223.

The assignment of the MODE parameter defines the operating mode of SFB 54 RALRM. With MODE = 0, the SFB shows you the component triggering the interrupt in the ID parameter; NEW is assigned TRUE. With MODE = 1, all output parameters are

written. With MODE = 2, you check whether the component specified by the F_ID parameter was the one triggering the interrupt. If this is the case, the NEW parameter has the value TRUE, and all other output parameters are written.

In order to work correctly, the SFB 54 RALRM requires its own instance data for each call in the various organization blocks, e.g. an own instance data block in each case.

22 Start-up Characteristics

22.1 General Remarks

22.1.1 Operating Modes

Before the CPU begins processing the main program following power-up, it executes a start-up routine. START-UP is one of the CPU's operating modes, as is STOP or RUN. This chapter describes the CPU's activities on a transition from and to START-UP and in the restart routine itself.

Following power-up ①, the CPU is in the STOP mode (Figure 22.1). If the mode selector on the CPU's front panel is at RUN or RUN-P, the CPU switches to START-UP mode ②, then to RUN mode ③. If an "unrecoverable" error occurs while the CPU is in START-UP or RUN mode or if you position the mode selector to STOP, the CPU returns to the STOP mode ④ ⑤.

The user program is tested with breakpoints in single-step operation in the HOLD mode. You can switch to this mode from both RUN and START-UP, and return to the original mode when you abort the test ⑥ ⑦. You can also set the CPU to the STOP mode from the HOLD mode ⑧.

When you parameterize the CPU, you can define restart characteristics with the "Restart" tab such as the maximum permissible amount of time for the Ready signals from the modules following power-up or whether the CPU is to start up when the configuration data do not coincide with the actual configuration or in what mode the CPU restart is to be in.

SIMATIC S7 has three restart modes, namely *cold restart*, *warm restart* and *hot restart*. On a cold restart or warm restart, the main program is always processed from the beginning. A hot restart resumes the main program at the point of interruption, and "finishes" the cycle.

S7 CPUs supplied before 10/98 have warm restart and hot restart.

You can scan a program on a single-shot basis in START-UP mode. STEP 7 provides organization blocks OB 102 (cold restart), OB 100 (warm restart) and OB 101 (hot restart) expressly for this purpose. Sample applications are the parameterization of modules unless this was already taken care of by the CPU, and the programming of defaults for your main program.

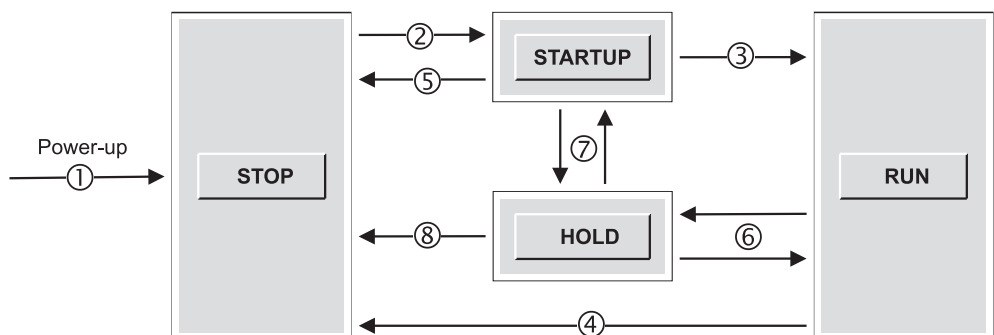


Figure 22.1 CPU Operating Modes

22.1.2 HOLD Mode

The CPU changes to the HOLD mode when you test the program with breakpoints (in “single-step mode”). The STOP LED then lights up and the RUN LED blinks.

In HOLD mode, the output modules are disabled. Writing to the modules affects the module memory, but does not switch the signal states “out” to the module outputs. The modules are not reenabled until you exit the HOLD mode.

In HOLD mode, everything having to do with timing is discontinued. This includes, for example, the processing of timers, clock memory and run-time meters, cycle time monitoring and minimum scan cycle time, and the servicing of time-of-day and time-delay interrupts. Exception: the real-time clock continues to function normally.

Every time the progression is made to the next statement in test mode, the timers for the duration of the single step run a little further, thus simulating a dynamic behavior similar to “normal” program scanning.

In HOLD mode, the CPU is capable of passive communication, that is, it can receive global data or take part in the unilateral exchange of data.

If the power fails while the CPU is in HOLD mode, battery-backed CPUs go to STOP on power recovery. CPUs without backup batteries execute an automatic warm restart.

22.1.3 Disabling the Output Modules

In the STOP and HOLD modes, modules are disabled (OD (output disable) signal). Disabled output modules output a zero signal or, if they have the capability, the replacement value. Via a variable table, you can control outputs on the modules with the “Isolate PQ” function, even in STOP mode.

During restart, the output modules remain disabled. Only when the cyclic scan begins are the output modules enabled.

On a cold restart (OB 102) and warm restart (OB 100), the process images and the module memory are cleared. If you want to scan inputs

in OB 102 or in OB 100, you must load the signal states from the module using direct access. You can then set the inputs (transfer them, for instance, with load statements or with the MOVE box from address area PI to address area I), then work with the inputs.

On a warm restart, the “old” process-image input and process-image output tables, which were valid prior to power-down or STOP, are used in OB 101 and in the remainder of the cycle. At the end of that cycle, the process-image output table is transferred to module memory (but not yet switched through to the external outputs, since the output modules are still disabled).

You now have the option of parameterizing the CPU to clear the process-image output table and the module memory at the end of the warm restart. Before switching to OB 1, the CPU revokes the Disable signal so that the signal states in the module memory are applied to the external outputs.

22.1.4 Restart Organization Blocks

On a cold restart, the CPU calls organization block OB 102; on a warm restart, it calls organization block OB 100. In the absence of OB 100 or OB 102, the CPU begins cyclic program execution immediately.

On a warm restart, the CPU calls organization block OB 101 on a single-shot basis before processing the main program. If there is no OB 101, the CPU begins scanning at the point of interruption.

The start information in the temporary local data has the same format for the restart organization blocks; Table 22.1 shows the start information for OB 100. The reason for the restart is shown in the restart request (Byte 1):

- B#16#81 Manual warm restart (OB 100)
- B#16#82 Automatic warm restart (OB 100)
- B#16#83 Manual hot restart (OB 101)
- B#16#84 Automatic hot restart (OB 101)
- B#16#85 Manual cold restart (OB 102)
- B#16#86 Automatic cold restart (OB 102)

Table 22.1 Start Information for the Restart OBs

Byte	Name	Data Type	Description	Contents
0	OB100_EV_CLASS	BYTE	Event class	B#16#13
1	OB100_STRTUP	BYTE	Restart request (see text)	B#16#8x (see text)
2	OB100_PRIORITY	BYTE	Priority class	Default value 27
3	OB100_OB_NUMBR	BYTE	OB number	100, 101 or 102
4	OB100_RESERVED_1	BYTE	Reserved	-
5	OB100_RESERVED_2	BYTE	Reserved	-
6..7	OB100_STOP	WORD	Number of the stop event	(see Instruction Manual)
8..11	OB100_STRT_INFO	DWORD	Additional information on the current restart	(see Instruction Manual)
12..19	OB100_DATE_TIME	DT	Date and time event occurred	Call time of the OB

The number of the stop event and the additional information define the restart more precisely (tells you, for example, whether a manual warm restart was initiated via the mode selector). With this information, you can develop an appropriate event-related restart routine.

Note that no asynchronous system blocks can be processed in the startup program of an S7-300 CPU. You can set or reset outputs in the process image in the startup program, but the transmission to the output modules only takes place when transferring to RUN mode.

22.2 Power-Up

22.2.1 STOP Mode

The CPU goes to STOP in the following instances:

- ▷ When the CPU is switched on
- ▷ When the mode selector is set from RUN to STOP
- ▷ When an “unrecoverable” error occurs during program scanning
- ▷ When system function SFC 46 STP is executed
- ▷ When requested by a communication function (stop request from the programming device or via communication function blocks from another CPU)

The CPU enters the reason for the STOP in the diagnostic buffer. In this mode, you can also read the CPU information with a programming device in order to localize the problem.

In STOP mode, the user program is not scanned. The CPU retrieves the settings – either the values which you entered in the hardware configuration data when you parameterized the CPU or the defaults – and sets the modules to the specified initial state.

In STOP mode, the CPU can receive global data via GD communication and carry out passive unilateral communication functions. The real-time clock keeps running.

You can parameterize the CPU in STOP mode, for instance you can also set the MPI address, transfer or modify the user program, and execute a CPU memory reset.

22.2.2 Memory Reset

A memory reset sets the CPU to the “initial state”. You can initiate a memory reset with a programming device only in STOP mode or with the mode selector: hold the switch in the MRES position for at least 3 seconds then release, and after a maximum of 3 seconds hold it in the MRES position again for at least 3 seconds.

The CPU erases the entire user program both in work memory and in RAM load memory. System memory (for instance bit memory, timers and counters) is also erased, regardless of retentivity settings. With a micro memory card, the

contents of the load memory are retained during a memory reset.

The CPU sets the parameters for all modules, including its own, to their default values. The MPI parameters of the first interface are an exception. They are not changed so that a CPU whose memory has been reset can still be addressed on the MPI bus. A memory reset also does not affect the diagnostic buffer, the real-time clock, or the run-time meters.

If a micro memory card or a memory card with Flash EPROM is inserted, the CPU copies the user program from the memory card to work memory. The CPU also copies any configuration data it finds on the memory card.

22.2.3 Restoring the factory settings

In the case of newer CPUs, you can restore the factory settings with “Reset to factory settings”. Proceed as follows:

- ▷ Switch off the supply voltage and remove the micro memory card or the memory card.
- ▷ Hold the mode selector switch in the MRES position and switch the supply voltage on again.
- ▷ If the LEDs SF (S7-300) or INTF (S7-400), FRCE, RUN and STOP flash slowly, release the mode selector switch again, set it to MRES again within 3 s, and hold it in this position.
- ▷ Wait until only the SF or INTF LED flashes. During this time period (approx. 5 s), you can abort the reset procedure by releasing the mode selector.
- ▷ If the SF or INTF LED shows a continuous light, release the mode selector switch.

The CPU starts up without battery backing and all LEDs light up. It executes a memory reset, then sets the MPI address to 2 and the MPI data rate to 187.5 kbit/s. As well as the memory reset, the real-time clock is set to the start date and the operating hours counter and diagnostics buffer are deleted. Following this, the CPU enters the event “Reset to factory settings” in the diagnostics buffer, and goes to STOP mode.

22.2.4 Retentivity

A memory area is retentive when its contents are retained even when the mains power is switched off as well as on a transition from STOP to RUN following power-up. In the case of the current S7-300 CPUs, retentivity is implemented with a micro memory card. In the case of the S7-400 CPUs, battery backup is required for retentivity.

Retentive memory areas may be those for bit memory, timers, counters and data blocks. The length of the retentive areas depends on the CPU. You can specify the number of retentive memory bytes, timers and counters via the “Retentivity” tab when you parameterize the CPU.

The contents of data blocks in the RAM can also be retentive. The available retentive area is CPU-specific. You can define the retentivity of a data block with the block property *Non-Retain* (see Chapter 3.2.3, “Block Properties”).

With the S7-300 with micro memory card, the bit memories, timers and counters set as retentive as well as the user program and user data are saved on the micro memory card, where they are retentive even without a battery backup. With a warm restart, the non-retentive bit memories, timers and counters are deleted. The contents of data blocks declared as “non-retentive” are initialized during a warm restart (loaded with the initial values from load memory) or set to zero if a load memory object is not present.

With the S7-400, a battery backup is required for retentivity. A cold restart deletes all address areas, and loads the user program and the (configured) user data from load memory into work memory. With a warm restart, the values of the bit memories, timers and counters set as retentive are retained; the user program and the user data remain unchanged.

22.2.5 Restart Parameterization

On the “Restart” tab of the CPUs, you can affect a restart with the following settings:

- ▷ Restart when the set configuration is not the same as the actual configuration

A restart is executed even if the parameterized hardware configuration does not agree with the actual configuration. Exception: the configured PROFIBUS DP interfaces must always be present and ready for operation.

- ▷ Reset outputs on hot restart
The S7-400 CPUs delete all process-image output tables and all peripheral outputs during a hot restart.
- ▷ Disable hot restart at manual restart
A hot restart based on manual input or communications job is not permissible.
- ▷ Restart following POWER UP
Definition of the type of restart following power up
- ▷ Monitoring time for ready signal of the modules
If the monitoring time for a module times out, it is considered to be non-existent. The response of the CPU then depends on the setting "Startup when set configuration not same as actual configuration". The result is entered in the diagnostics buffer. This monitoring time is important for switching on the power on expansion racks or distributed I/O.
- ▷ Monitoring time for transferring the parameters to the modules
If the monitoring time for a module times out, it is considered to be non-existent. The response of the CPU then depends on the setting "Startup when set configuration not same as actual configuration". The event is entered in the diagnostics buffer. (In the event of this error, you can only parameterize the CPU with a higher monitoring time – without memory reset – if you transfer the system data of an "empty" project in which the new value of the monitoring time is entered, so that the module parameterization is completed within the "old" monitoring time.)
- ▷ Monitoring time for hot restart
If the time between power off and power on or the time between STOP and RUN is greater than the monitoring time, a hot

restart is not carried out. The specification 0 ms switches the monitor off.

22.3 Types of Restart

22.3.1 START-UP Mode

The CPU executes a restart in the following cases:

- ▷ When the mains power is switched on
- ▷ When the mode selector is set (key lock switch: turn from STOP to RUN or RUN-P), or the toggle switch changed (switch from STOP to RUN)
- ▷ On a request from a communication function (initiated from a programming device or via communication function blocks from another CPU)

A *manual* restart is initiated via the mode selector or a communication function, an *automatic* restart by switching on the mains power.

The restart routine may be as long as required, and there is no time limit on its execution; the scan cycle monitor is not active.

During the execution of the restart routine, no interrupts will be serviced. Exceptions are errors that are handled as in RUN (call of the relevant error organization blocks).

In the restart routine, the CPU updates the timers, the run-time meters and the real-time clock.

During restart, the output modules are disabled, i.e., output signals cannot be transmitted. The output disable is only revoked at the end of the restart and prior to starting the cyclic program.

A restart routine can be aborted, for instance when the mode selector is actuated or when there is a power failure. The aborted restart routine is then executed from the beginning when the power is switched on. If a cold restart or warm start is aborted, it must be executed again. If a hot restart is aborted, all restart types are possible.

Figure 22.2 shows the activities carried out by an S7-400 CPU during a restart.

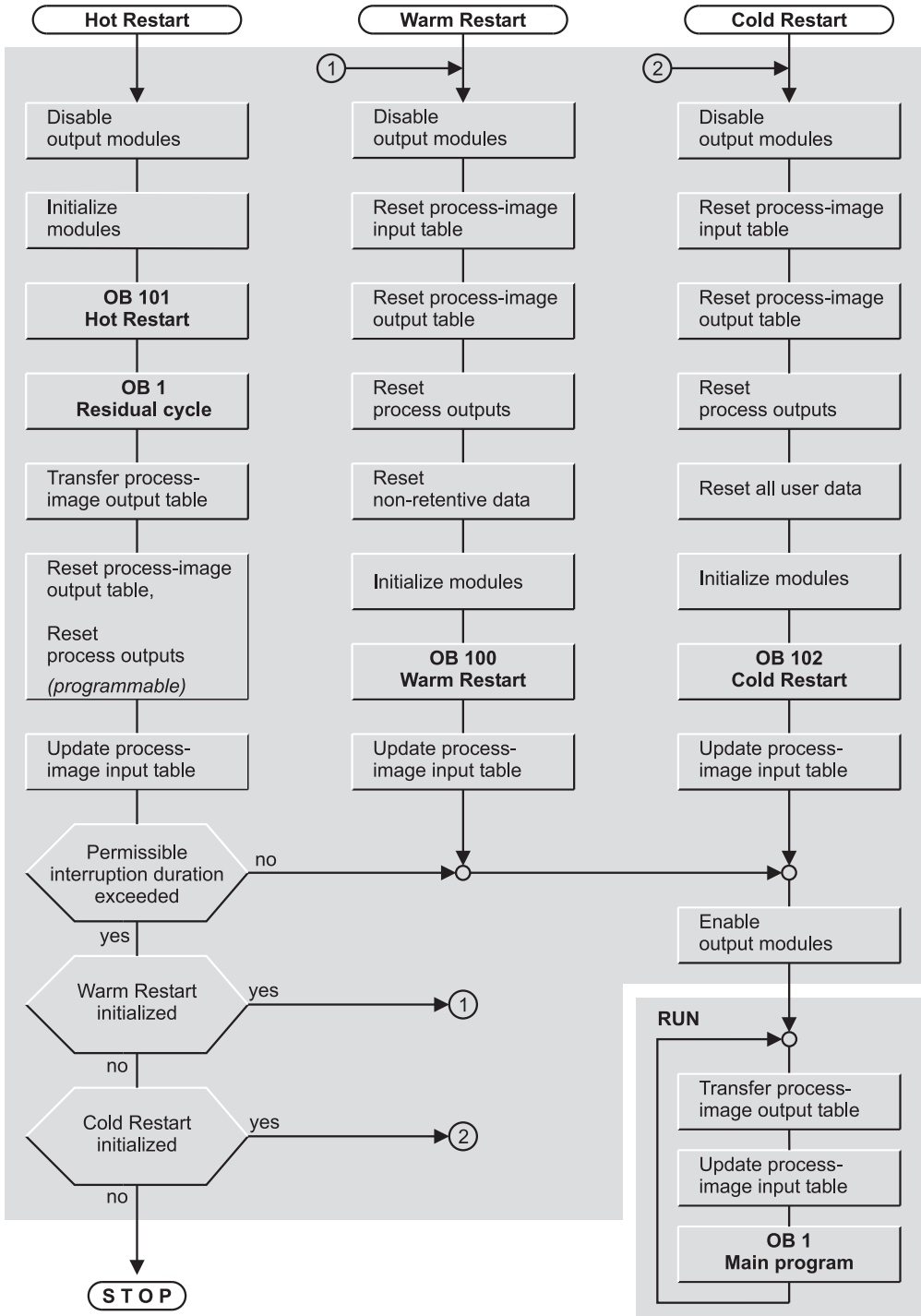


Figure 22.2 CPU Activities During Restart (S7-400)

22.3.2 Cold Restart

On a cold restart, the CPU sets both itself and the modules to the programmed initial state, deletes all data in the system memory (including the retentive data), calls OB 102, and then executes the main program in OB 1 from the beginning.

The current program and the current data in work memory are deleted and with them also the data blocks generated by a system function; the program from load memory is reloaded. (In contrast to memory reset, a RAM load memory is not deleted.)

Manual cold restart

With newer CPUs, a cold restart can no longer be triggered manually using the mode selector. With older CPUs, a manual cold restart is triggered using the mode selector if the switch is held in the MRES position for at least 3 s before transferring from STOP to RUN or RUN-P.

A manual cold restart can also be triggered by a communications function from a programming device or by a system block from another CPU. In this case, the mode switch must be at RUN or RUN-P.

A manual cold restart can always be initiated unless the CPU requests a memory reset.

Automatic cold restart

An automatic cold restart is initiated by switching on the mains power. The cold restart is executed if

- ▷ the CPU was not at STOP when the power was switched off
- ▷ the mode selector is at RUN or RUN-P
- ▷ the CPU was interrupted by a power outage while executing a cold restart
- ▷ “Cold restart” on “start following POWER UP” is parameterized

When operated without a backup battery, the CPU executes an automatic non-retentive warm restart. The CPU starts the memory reset automatically, then copies the user program

from the memory card to work memory. The memory card must be a Flash EPROM.

22.3.3 Warm Restart

On a warm restart, the CPU sets both itself and the modules to the programmed initial state, erases the non-retentive data in the system memory, calls OB 100, and then executes the main program in OB 1 from the beginning.

The current program and the data set as retentive in work memory are retained, as are the data blocks created by SFC.

Manual warm restart

A manual warm restart is initiated in the following instances:

- ▷ Via the mode selector on the CPU on a transition from STOP to RUN or RUN-P (on S7-400 CPUs with restart type switch, this is in the CRST position)
- ▷ Via a communication function from a PG or with an SFB from another CPU; the mode selector must be in the RUN or RUN-P position.

A manual warm restart can always be initiated unless the CPU requests a memory reset.

Automatic warm restart

An automatic warm restart is initiated by switching on the mains power. The warm restart is executed if

- ▷ the CPU was not at STOP when the power was switched off
- ▷ the mode selector is at RUN or RUN-P
- ▷ the CPU was interrupted by a power outage while executing a warm restart
- ▷ “Complete restart (warm restart)” on “start following POWER UP” is parameterized

If there is a restart type switch, it remains without effect in the case of automatic warm restart.

If the CPU contains a micro memory card, it reacts exactly like a CPU with backup battery. When operated without a micro memory card and without a backup battery, the CPU executes an automatic non-retentive warm restart. The

CPU starts the memory reset automatically, then copies the user program from the memory card to work memory. The memory card must be a Flash EPROM.

22.3.4 Hot Restart

A hot restart is possible only on an S7-400.

On a STOP or power outage, the CPU saves all interrupts as well as the internal CPU registers that are important to the processing of the user program. On a hot restart, it can therefore resume at the location in the program at which the interruption occurred. This may be the main program, or it may be an interrupt or error handling routine. All (“old”) interrupts are saved and will be serviced.

The so-called “residual cycle”, which extends from the point at which the CPU resumes the program following a hot restart to the end of the main program, counts as part of the restart. No (new) interrupts are serviced. The output modules are disabled, and are in their initial state.

A hot restart is permitted only when there have been no changes in the user program while the CPU was at STOP, such as modification of a block.

By parameterizing the CPU accordingly, you can specify how long the interruption may be for the CPU to still be able to execute a warm restart (from 100 milliseconds to 1 hour). If the interruption is longer, only a cold or warm restart is allowed. The length of the interruption is the amount of time between exiting of the RUN mode (STOP or power-down) and reentry into the RUN mode (following execution of OB 101 and the residual cycle).

Manual hot restart

A manual hot restart is initiated

- ▷ If the mode selector was at RUN or RUN-P when the CPU was switched on by moving the mode selector from STOP to RUN or RUN-P when the restart switch is at WRST (only possible on CPUs with restart type switch)

- ▷ Via a communications function from a programming device or with a system block (SFB) from another CPU; the mode selector must be at RUN or RUN-P.

A manual hot restart is possible only when the hot restart disable was revoked in the “Restart” tab when the CPU was parameterized. The cause of the STOP must have been a manual activity, either via the mode selector or through a communication function; only then can a manual warm restart be executed while the CPU is at STOP.

Automatic hot restart

An automatic hot restart is initiated by switching on the mains power. The CPU executes an automatic hot restart only in the following instances:

- ▷ If it was not at STOP when switched off
- ▷ If the mode selector was at RUN or RUN-P when the CPU was switched on
- ▷ “Hot restart” on “start following POWER UP” is parameterized
- ▷ If the backup battery is inserted and in working order

The position of the restart switch is irrelevant to an automatic hot restart.

22.4 Ascertaining a Module Address

Signal modules, or to be more exact the user data on input/output modules, are addressed in two manners: you use the *logical address* in the user program to address the inputs and outputs. This corresponds to the absolute address, and can be made easier to read by using symbols. The smallest logical addresses is the base address or module starting address. The CPU uses the *geographical address* to address the modules. You require the geographical address if you wish to find out the slot number of the module. This applies similarly to the user data on stations of the distributed I/O.

You can use the following system blocks to ascertain the geographical address from the logical address and vice versa:

- ▷ SFC 70 GEO_LOG
Ascertain logical base address
- ▷ SFC 5 GADR_LGC
Ascertain logical address of a module channel
- ▷ SFC 50 RD_LGADR
Ascertain all logical addresses of a module
- ▷ SFC 71 LOG_GEO
Ascertain geographical address
- ▷ SFC 49 LGC_GADR
Ascertain slot address of a module

Table 22.2 shows the parameters for these system blocks.

The SFC 5 GADR_LGC, 49 LGC_GADR and 50 RD_LGADR have IOID and LADDR as common parameters for the logical address (= address in the I/O area). IOID is either B#16#54, which stands for the peripheral inputs (PIs) or B#16#55, which stands for the peripheral outputs (PQs). LADDR contains an I/O address in the PI or PQ area which corresponds to the specified channel. If the channel is 0, it is the module start address.

With the SFCs 70 GEO_LOG and 71 LOG_GEO, the logical address is on its own in the LADDR parameter. A differentiation is made in bit 15 as to whether the address is assigned to an input (= 0) or an output (= 1).

For the addresses determined using these system blocks, an assignment must have been made using the Hardware Configuration tool between logical address (module starting address) and slot address (location of module in a rack or in a station of the distributed I/O).

SFC 70 LOG_GEO

Ascertain logical start address

System function SFC 70 LOG_GEO returns the logical address of a module or station. The assignment of the MASTER parameter indicates whether the station or module is inserted in a rack (central design) or whether the station is operated in a PROFIBUS or PROFINET system. Use the SLOT parameter to specify the slot number in the rack or station, and the SUBSLOT parameter for the number of the submodule. The LADDR parameter then returns the

base address of the submodule. With SUBSLOT = 0, the diagnostics address of the module or station is returned.

SFC 70 GEO_LOG replaces the SFC 5 GADR_LGC, and can be used in conjunction with PROFINET IO.

SFC 5 GADR_LGC

Ascertain the logical address of a module channel

System function SFC 5 GADR_LGC returns the logical address of a channel when you specify the slot address ("geographical" address). Enter the number of the DP master system ID in the SUBNETID parameter if the module belongs to the distributed I/O or B#16#00 if the module is plugged into the central rack or an expansion rack. The RACK parameter specifies the number of the rack or, in the case of distributed I/O, the number of the station. If the module has no submodule slot, enter B#16#00 in the SUBSLOT parameter. SUBADDR contains the address offset in the module's user data (W#16#0000, for example, stands for the module start address).

SFC 71 GEO_LOG

Ascertain geographical address

System function SFC 71 GEO_LOG returns the geographical address of a module or station if you define the logical base address. The value in the AREA parameter defines the system in which the module is used (Table 22.3).

SFC 71 GEO_LOG replaces the SFC 49 LGC_GADR and can also be used in association with PROFINET IO.

SFC 49 LGC_GADR

Ascertain the slot address of a module

SFC 49 LGC_GADR returns the slot address of a module when you specify an arbitrary logical module address. Subtracting the address offset (parameter SUBADDR) from the specified user data address gives you the module starting address. The value in the AREA parameter specifies the system in which the module is operated (Table 22.3).

Table 22.2 Parameters for the System Blocks Used to Ascertain the Module Address

SFC	Parameter	Declaration	Data Type	Contents, Description
5	SUBNETID	INPUT	BYTE	Area identifier
	RACK	INPUT	WORD	Number of the rack
	SLOT	INPUT	WORD	Number of the slot
	SUBSLOT	INPUT	BYTE	Number of the submodule
	SUBADDR	INPUT	WORD	Offset in the module's user data address area
	RET_VAL	RETURN	INT	Error information
	IOID	OUTPUT	BYTE	Area identifier
	LADDR	OUTPUT	WORD	Logical address of the channel
50	IOID	INPUT	BYTE	Area identifier
	LADDR	INPUT	WORD	A logical module address
	RET_VAL	RETURN	INT	Error information
	PEADDR	OUTPUT	ANY	WORD field for the PI addresses
	PECOUNT	OUTPUT	INT	Number of PI addresses returned
	PAADDR	OUTPUT	ANY	WORD field for the PQ addresses
	PACOUNT	OUTPUT	INT	Number of PQ addresses returned
49	IOID	INPUT	BYTE	Area identifier
	LADDR	INPUT	WORD	A logical module address
	RET_VAL	RETURN	INT	Error information
	AREA	OUTPUT	BYTE	Area identifier
	RACK	OUTPUT	WORD	Number of the rack
	SLOT	OUTPUT	WORD	Number of the slot
	SUBADDR	OUTPUT	WORD	Offset in the module's user data address area
70	MASTER	INPUT	INT	Master system ID 0 = central I/O 1 to 31 = PROFIBUS DP 100 to 115 = PROFINET IO
	STATION	INPUT	INT	Station number; with central I/O: rack number
	SLOT	INPUT	INT	Number of the slot
	SUBSLOT	INPUT	INT	Number of the submodule
	RET_VAL	RETURN	INT	Error information
	LADDR	OUTPUT	WORD	Base address of station or module
71	LADDR	INPUT	WORD	Base address of station or module
	RET_VAL	RETURN	INT	Error information
	AREA	OUTPUT	INT	Area identifier (see Table 22.3)
	MASTER	OUTPUT	INT	Master system ID 0 = central I/O 1 to 31 = PROFIBUS DP 100 to 115 = PROFINET IO
	STATION	OUTPUT	INT	Station number; with central I/O: rack number
	SLOT	OUTPUT	INT	Number of the slot
	SUBSLOT	OUTPUT	INT	Number of the submodule
	OFFSET	OUTPUT	INT	Offset in address space of module/submodule

Table 22.3 Meaning of Output Parameters of SFC 49 LGC_GADR and SFC 71 LOG_GEO

AREA	System	Meaning of output parameters of SFC 49 LGC_GADR	Meaning of output parameters of SFC 71 LOG_GEO
0	S7-400	RACK = rack number SLOT = slot number SUBADDR = difference from base address	MASTER = 0 STATION = rack number SLOT = slot number SUBSLOT = 0 OFFSET = difference from base address
1	S7-300		
2	Distributed I/O	RACK Low byte = station number High byte = DP master system ID SLOT = slot number SUBADDR = difference from base address	With PROFIBUS DP: MASTER = DP master system ID STATION = station number SLOT = slot number SUBSLOT = 0 OFFSET = difference from base address With PROFINET IO: MASTER = PROFINET IO system ID STATION = station number SLOT = slot number SUBSLOT = subslot number OFFSET = difference from base address
3	S5 P area	RACK = rack number SLOT = slot number of adapter casing SUBADDR = address in S5 area	MASTER = 0 STATION = rack number SLOT = slot number of adapter casing SUBSLOT = 0 OFFSET = address in S5 area
4	S5 Q area		
5	S5 IM3 area		
6	S5 IM4 area		

SFC 50 RD_LGADR**Ascertain all logical addresses for a module**

SFC 50 RD_LGADR returns all logical addresses for a module when you specify an arbitrary address from the user data area.

Use the PEADDR and PAADDR parameters to define an area of WORD components (a word-based ANY pointer, for example P#DBzD-BXy x WORD nnn).

SFC 50 then shows the number of entries returned in these areas in the RECOUNT and PACOUNT parameters.

deviate from the default. To specify parameters, open the module in the hardware configuration and fill in the tabs in the dialog box. When you transfer the *System Data* object in the *Blocks* container to the PLC, you are also transferring the module parameters.

The CPU transfers the module parameters to the module automatically in the following cases

- ▷ On restart
- ▷ When a module has been plugged into a configured slot (S7-400)
- ▷ Following the “return” of a rack or a distributed I/O station.

22.5 Parameterizing Modules**22.5.1 General remarks on parameterizing modules**

Most S7 modules can be parameterized, that is to say, values may be set on the module which

Static and dynamic module parameters

The module parameters are divided into static parameters and dynamic parameters. You can set both parameter types offline in the Hard-

ware Configuration. You can also modify the dynamic parameters at runtime by calling a system block. In the restart routine, the parameters set on the modules using system blocks are overwritten by the parameters set (and stored on the CPU) via the Hardware Configuration.

The parameters for the signal modules are in two data records: the static parameters in data record 0 and the dynamic parameters in data record 1. You can transfer both data records to the module with system function SFC 57 PARM_MOD, data record 0 or 1 with system function SFC 56 WR_DPARM, and only data record 1 with system function SFC 55 WR_PARM. The data records must be in the system data blocks on the CPU.

After parameterization of an S7-400 module, the specified values do not go into force until bit 2 (“Operating mode”) in byte 2 of diagnostic data record has assumed the value “RUN”. The diagnostic data record can be read with system function SFC 59 RD_REC or system function block SFB 52 RDREC.

Asynchronous processing of system blocks

Apart from the system function SFC 54 RD_DPARM, the system blocks for module parameterization and transmission of records work asynchronously. Execution of the function covers several calls, and is triggered by the block parameter REQ = “1”. During processing of the job, the BUSY parameter is at “1”, and the error information has the value W#16#7001 (job being processed). The error information is in the RET_VAL parameter for the system functions, and in bytes 2 and 3 of the STATUS parameter for the system function blocks.

A certain job for a module is specified by the module starting address and the data record number. As long as BUSY = “1”, a renewed call for the same job with REQ = “1” has no effect, and the error information is set to W#16#7002.

If an error occurs when triggering an job, this is signaled by the error information, and BUSY remains “0”.

If the job has been completed, BUSY has the status “0”. If the job is completed without error, the error information has the value W#16#0000; with the system function SFC 59

RD_REC, the number of transmitted bytes is present in RET_VAL. In the event of an error, the error code is present in the error information.

Module and data record addressing

As far as addressing for data transfer is concerned, use the module start address. With mixed modules having input and output areas, use the lower area start address. If you assigned the same start address to both the input and output areas, use the identifier for an input address. Use the I/O identifier regardless of whether you want to execute a read or write operation.

The module start address is parameterized either using the IOID and LADDR parameters or – with newer system blocks – using the LADDR on its own. In this case, bit 15 determines whether an input (“0”) or output (“1”) is involved. (With the system function blocks SFB 52 RDREC and SFB 53 WRREC, this is the parameter ID.)

Enter the data record number in the RECNUM or INDEX parameter.

Use the RECORD parameter with the data type ANY to define an area of BYTE components. This may be a variable of type ARRAY, STRUCT or UDT, or an ANY pointer of type BYTE (for example P#DBzDBXy.x BYTE *nnn*). If you use a variable, it must be a “complete” variable; individual array or structure components are not permissible.

Permissible data record numbers

Data records with the numbers 1 to 240 are permissible for the system functions for module parameterization. The specified data records must be present in the system data in the case of system blocks SFC 54 RD_DPARM, SFC 56 WR_DPARM and SFB 81 RD_DPAR.

System function SFC 58 WR_REC can process data records in the range 2 to 240, SFC 59 RD_REC in the range 0 to 240. System function blocks SFB 52 RDREC and SFB 53 WRREC transmit data records with numbers 0 to 255.

Data records 0 and 1 have a special significance with SIMATIC S7:

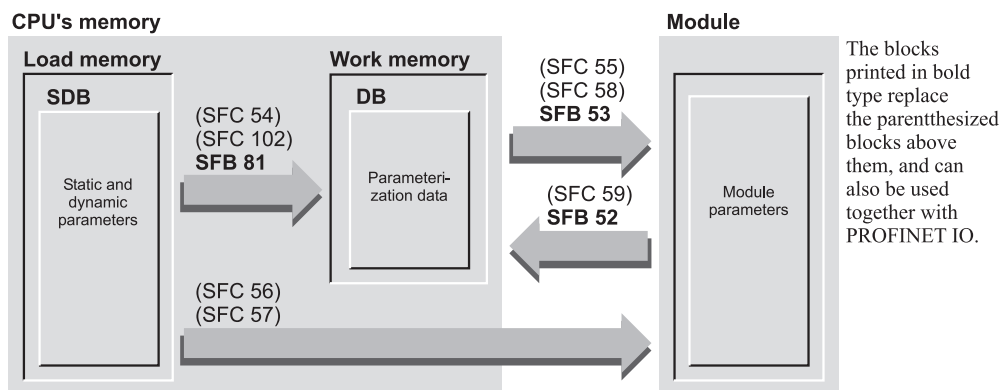


Figure 22.3 System Blocks for Module Parameterization

- ▷ Data record 0: reading of diagnostics data (4 bytes) and writing of static module parameters
- ▷ Data record 1: reading of diagnostics data (data record 0 and further data) and writing of dynamic module parameters

A data record can be up to 240 bytes long.

Module parameterization with PROFINET IO

Connection of distributed I/O over PROFINET IO requires an extended quantity framework for the module parameterization compared to PROFIBUS DP, and new system blocks are provided for this. These new system blocks can replace the previous blocks. Figure 22.3 provides an overview of the system blocks for module parameterization.

22.5.2 System Blocks for Module Parameterization

The following system blocks are available for parameterizing modules:

- ▷ SFB 81 RD_DPAR
Read predefined parameters
- ▷ SFC 54 RD_DPARM
Read predefined parameters
- ▷ SFC 55 WR_PARM
Write dynamic parameters
- ▷ SFC 56 WR_DPARM
Write predefined parameters

- ▷ SFC 57 PARM_MOD
Parameterize module

- ▷ SFC 102 RD_DPARA
Read predefined parameters

The parameters for the listed system blocks are described in Table 22.4.

SFB 81 RD_DPAR

Reading predefined parameters

System function block SFB 81 RD_DPAR transmits the data record with the number specified in the INDEX parameter from the corresponding SDB system data block to the destination area specified at the RECORD parameter.

The transmission is carried out in asynchronous mode, and can be divided between several program cycles; the BUSY parameter is “1” during the transmission. Following successful transmission, the VALID parameter is “1”, and the LEN parameter contains the number of transmitted data bytes.

The read data record can now be evaluated or modified, for example, and written into the module using SFB 53 WRREC.

SFB 81 RD_DPAR replaces SFC 102 RD_DPARA and SFC 54 RD_PARM.

SFC 102 RD_DPARA

Reading predefined parameters

System function SFC 102 RD_DPARA transmits the data record with the number specified in the RECNUM parameter from the corre-

Table 22.4 Parameters of System Blocks for Module Parameterization

Present in SFC				Parameter	Declaration	Data Type	Contents, Description
-	55	56	57	REQ	INPUT	BOOL	“1” = write request
54	55	56	57	IOID	INPUT	BYTE	B#16#54 = peripheral inputs PI B#16#55 = peripheral outputs PQ
54	55	56	57	LADDR	INPUT	WORD	Module start address
54	55	56	-	RECNUM	INPUT	BYTE	Data record number
-	55	-	-	RECORD	INPUT	ANY	Source area for data record
54	55	56	57	RET_VAL	RETURN	INT	Error information
-	55	56	57	BUSY	OUTPUT	BOOL	Transfer still in progress if “1”
54	-	-	-	RECORD	OUTPUT	ANY	Destination area for data record

Present in		Parameter	Declaration	Data Type	Contents, Description
SFC 102	SFB 81	REQ	INPUT	BOOL	“1” = write request
SFC 102	SFB 81	LADDR	INPUT	WORD	Module start address
SFC 102	-	RECNUM	INPUT	BYTE	Data record number
-	SFB 81	INDEX	INPUT	INT	Data record number
SFC 102	-	RET_VAL	RETURN	INT	Error information
-	SFB 81	VALID	OUTPUT	BOOL	New data record received and valid
SFC 102	SFB 81	BUSY	OUTPUT	BOOL	Transfer still in progress if “1”
-	SFB 81	ERROR	OUTPUT	BOOL	Error has occurred if “1”
-	SFB 81	STATUS	OUTPUT	DWORD	Call identification or error information (bytes 2 and 3)
-	SFB 81	LEN	OUTPUT	INT	Length of read data
SFC 102	-	RECORD	OUTPUT	ANY	Destination area for data record
-	SFB 81	RECORD	IN_OUT	ANY	Destination area for data record

sponding SDB system data block to the destination area specified at the RECORD.

The transmission is carried out in asynchronous mode, and can be divided between several program cycles; the BUSY parameter is “1” during the transmission.

SFC 102 replaces the SFC 54 RD_DPARM which works in synchronous mode.

SFC 54 RD_DPARM

Reading predefined parameters

System function SFC 54 RD_DPARM transfers the data record with the number specified in the RECNUM parameter from the relevant SDB system data block to the destination area specified at the RECORD parameter.

The transmission is carried out in asynchronous mode; the system function is processed until the data record has been transmitted. Since reading is from load memory, the relatively long processing time with large data records may – depending on the application – be considered as inconvenient. In this case, use the SFB 81 RD_DPAR or the SFC 102 RD_DPARA, which execute this function in asynchronous mode.

You can now, for example, evaluate or modify this read data record and write it to the module with SFB 53 WRREC or SFC 58 WR_REC.

SFC 55 WR_PARM

Writing dynamic parameters

System function SFC 55 WR_PARM transfers the data record addressed by RECORD to the

module specified by the IOID and LADDR parameters. Specify the number of the data record in the RECNUM parameter. The data record may only contain the dynamic module parameters, it must not be data record 0. If the module parameters are present in the associated SDB, they must not be identified as static.

When the job is initiated, the SFC reads the entire data record; the transfer may be distributed over several program scan cycles. The BUSY parameter is “1” during the transfer.

SFC 56 WR_DPARM

Writing predefined parameters

System function SFC 56 WR_DPARM transfers the data record with the number specified in the RECNUM parameter from the relevant SDB system data block to the module identified by the IOID and LADDR parameters.

The transfer may be distributed over several program scan cycles; the BUSY parameter is “1” during the transfer.

SFC 57 PARM_MOD

Parameterizing a module

System function SFC 57 PARM_MOD transfers all the data records programmed when the module was parameterized via the Hardware Configuration.

The transfer may be distributed over several program scan cycles; the BUSY parameter is “1” during the transfer.

22.5.3 Blocks for Transmitting Data Records

The following system blocks are available for transmitting data records:

- ▷ SFB 52 RDREC
Read data record
- ▷ SFC 59 RD_REC
Read data record
- ▷ SFB 53 WRREC
Write data record
- ▷ SFC 58 WR_REC
Write data record

The parameters of the listed system functions are described in Table 22.5 and those of the system function blocks in Table 22.6.

With an S7-300 CPU, you can process up to four write jobs and four read jobs simultaneously per DP segment. With an S7-400 CPU, up to eight write jobs and eight read jobs can be simultaneously active per DP segment. You can simultaneously execute a maximum total of 32 write jobs and 32 read jobs on external DP segments.

SFB 52 RDREC

Reading a data record

System function block SFB 52 RDREC with “1” at the REQ parameter reads the data record INDEX from the module and stores it in the destination area RECORD. The destination area must be longer than or at least as long as the data record. Use the MLEN parameter to specify how many bytes you wish to read.

Table 22.5 Parameters for System Functions Used for Data Transfer

Present in		Parameter	Declaration	Data Type	Contents, Description
SFC 58	SFC 59	REQ	INPUT	BOOL	“1” = write request
SFC 58	SFC 59	IOID	INPUT	BYTE	B#16#54 = input module B#16#55 = output module
SFC 58	SFC 59	LADDR	INPUT	WORD	Module start address
SFC 58	SFC 59	RECNUM	INPUT	BYTE	Data record number
SFC 58	-	RECORD	INPUT	ANY	Data record
SFC 58	SFC 59	RET_VAL	RETURN	INT	Error information
SFC 58	SFC 59	BUSY	OUTPUT	BOOL	Transfer still in progress if “1”
-	SFC 59	RECORD	OUTPUT	ANY	Data record

Table 22.6 Parameters for System Function Blocks Used for Data Transfer

Present in		Parameter	Declaration	Data Type	Contents, Description
SFB 52	SFB 53	REQ	INPUT	BOOL	“1” = write request
SFB 52	SFB 53	ID	INPUT	DWORD	Module start address Bit 15 = “0”: Input address Bit 15 = “1”: Output address
SFB 52	SFB 53	INDEX	INPUT	INT	Data record number
SFB 52	-	MLEN	INPUT	INT	Maximum number of bytes in data record to be read
-	SFB 53	LEN	INPUT	INT	Maximum number of bytes in data record to be sent
SFB 52	-	VALID	OUTPUT	BOOL	“1” = new data record has been received and is valid
-	SFB 53	DONE	OUTPUT	BOOL	Data record has been sent
SFB 52	SFB 53	BUSY	OUTPUT	BOOL	Transfer still in progress if “1”
SFB 52	SFB 53	ERROR	OUTPUT	BOOL	Error has occurred if “1”
SFB 52	SFB 53	STATUS	OUTPUT	DWORD	Status codes
SFB 52	-	LEN	OUTPUT	INT	Number of bytes of read data
SFB 52	SFB 53	RECORD	OUTPUT	ANY	Data record

The transfer may be distributed over several program scan cycles. The BUSY parameter is “1” during the transfer.

A signal status “1” at the VALID parameter signals that the data record has been read without faults. The LEN parameter then indicates the number of transmitted bytes.

In the event of an error, the ERROR parameter is set to “1” and the error information output in the STATUS parameter.

The system function block SFB 52 RDREC contains the functionality of system function SFC 59 RD_REC, and can replace the latter.

SFC 59 RD_REC Reading a data record

When the REQ parameter is “1”, SFC 59 RD_REC reads the data record addressed by the RECNUM parameter from the module and places it in destination area RECORD. The destination area must be longer than or at least as long as the data record. If the transfer is completed without error, the RET_VAL parameter contains the number of bytes transferred.

The transfer may be distributed over several program scan cycles; the BUSY parameter is “1” during the transfer.

S7-300 delivered prior to February 1997: the SFC reads as much data from the specified data record as the destination area can accommodate. The size of the destination area may not exceed that of the data record.

SFB 53 WRREC Writing a data record

System function block SFB 53 WRREC with “1” at the REQ parameter writes the data record INDEX from the source area RECORD to the module. Use the LEN parameter to specify how many bytes you wish to write.

The transfer may be distributed over several program scan cycles. The BUSY parameter is “1” during the transfer.

A signal status “1” at the DONE parameter signals that the data record has been written without faults. In the event of an error, the ERROR parameter is set to “1” and the error information output in the STATUS parameter.

The system function block SFB 53 WRREC contains the functionality of system function SFC 58 WR_REC, and can replace the latter.

SFC 58 WR_REC

Writing a data record

SFC 58 WR_REC transfers the data record addressed by the RECORD parameter and the

number RECNUM to the module defined by the IOID and LADDR parameters. A “1” in the REQ parameter starts the transfer. When the job is initiated, the SFC reads the complete data record.

The transfer may be distributed over several program scan cycles; the BUSY parameter is “1” during the transfer.

23 Error Handling

The CPU reports errors or faults detected by the modules or by the CPU itself in different ways:

- ▷ Errors in arithmetic operations (overflow, invalid REAL number) by setting status bits (status bit OV, for example, for a numerical overflow)
- ▷ Errors detected while executing the user program (synchronous errors) by calling organization blocks OB 121 and OB 122
- ▷ Errors in the programmable controller which do not relate to program scanning (asynchronous errors) by calling organization blocks OB 80 to OB 87

The CPU signals the occurrence of an error or fault, and in some cases the cause, by setting error LEDs on the front panel. In the case of unrecoverable errors (such as invalid OP code), the CPU goes directly to STOP.

With the CPU in STOP mode, you can use a programming device and the CPU information functions to read out the contents of the block stack (B stack), the interrupt stack (I stack) and the local data stack (L stack) and then draw conclusions as to the cause of error.

The system diagnostics can detect errors/faults on the modules, and enters these errors in a diagnostic buffer. Information on CPU mode transitions (such as the reasons for a STOP) are also placed in the diagnostic buffer.

The contents of this buffer are retained on STOP, on a memory reset, and on power failure, and can be read out following power recovery and execution of a start-up routine using a programming device.

On the new CPUs, you can use CPU parameterization to set the number of entries the diagnostics buffer is to hold.

23.1 Synchronous Errors

The CPU's operating system generates a synchronous error when an error occurs in immediate conjunction with program scanning. A distinction is made between two error types:

A **programming error** is present if execution of the program is faulty. Such errors include BCD conversion errors, errors with indirect addressing; addressing of missing timers, counters or blocks. Organization block OB 121 is called in the event of a programming error.

An **access error** (PZF) is present if an attempt is made to access a faulty or non-existent module or an I/O address not known on the CPU. The operating system reacts in different manners depending on the type of access:

- ▷ The I/O access is from the user program. In this case, the I/O access error organization block OB 122 is called.
- ▷ The PZF occurs during automatic updating of a (partial) process image. With S7-300 CPUs the default response is that an entry is not made into the diagnostics buffer and an OB is not called; S7-400 CPUs enter each PZF into the diagnostics buffer and start the OB 85. With newer CPUs, the response to a PZF can be parameterized (see "Program execution errors OB 85" in Chapter 23.3, "Asynchronous Errors").
- ▷ The PZF occurs if a partial process image is updated by a system function. The error and the address of the first byte signaling the error are then returned in their parameters (system functions SFC 26 UPDAT_PI, SFC 27 UPDAT_PO, SFC 126 SYNC_PI and SFC 127 SYNC_PO).

If the corresponding organization block OB 121 or OB 122 is not present when a synchronous error event occurs, the CPU enters the STOP status.

Table 23.1 Start Information for the Synchronous Error OBs 121 and 122

Byte	Variable Name	Data Type	Description, Contents	
0	OB12x_EV_CLASS	BYTE	B#16#25 = Call programming error OB 121 B#16#29 = Call access error OB 122	
1	OB12x_SW_FLT	BYTE	Error code (see Chapter 23.2.1, “Error Filters”)	
2	OB12x_PRIORITY	BYTE	Priority class in which the error occurred	
3	OB12x_OB_NUMBR	BYTE	OB number (B#16#79 or B#16#80)	
4	OB12x_BLK_TYPE	BYTE	Type of block interrupted (S7-400 only) OB: B#16#88, DB: B#16#8A, FB: B#16#8E, FC: B#16#8C	
5	OB121_RESERVED_1 OB122_MEM_AREA	BYTE	Byte assignment (B#16#xy):	
			7... (x) ... 4	3 ... (y) ... 0
			0 Bit access	0 I/O area PI or PQ
			1 Byte access	1 Process-image input table I
			2 Word access	2 Process-image output table Q
3 Doubleword access				
6...7	OB121_FLT_REG	WORD	OB 121: Error source: ▷ Errored address (at read/write access) ▷ Errored area (in the case of area error) ▷ Incorrect number of the block, timer/counter function	
	OB122_MEM_ADDR		OB 122: Address at which the error occurred	
8...9	OB12x_BLK_NUM	WORD	Number of the block in which the error occurred (S7-400 only)	
10...11	OB12x_PRG_ADDR	WORD	Error address in the block that caused the error (S7-400 only)	
12...19	OB12x_DATE_TIME	DT	Time at which programming error was detected	

Table 23.1 shows the start information for both synchronous error organization blocks.

The S7-400 CPUs distinguish between two types of access error: access to a non-existent module and invalid access attempt to an existing module (acknowledgment delay QVZ). If a module fails during operation, that module is marked “non-existent” approximately 150 μ s after an access attempt, and an I/O access error (PZF) is reported on every subsequent attempt to access the module. The CPU also reports an I/O access error when an attempt is made to access a non-existent module, regardless of whether the attempt was direct (via the I/O area) or indirect (via the process image).

If an access error occurs when writing the peripheral outputs, an S7-400 CPU tracks the output process image, an S7-300 CPU does not.

A synchronous error OB has the same priority (class) as the block in which the error occurred. The accumulators and address registers of asynchronous error OB contain the values present in

the block causing the error at the time of aborting. The data block registers are deleted; the condition code word has an undefined content.

Note that when a synchronous error OB is called, its 20 bytes of start information are also pushed onto the L stack for the priority class that caused the error, as are the other temporary local data for the synchronous error OB and for all blocks called in this OB. The area reserved for the temporary local data must be designed for this in every associated priority class (program execution level); (fixed setting with S7-300 CPUs, adjustable with S7-400 CPUs during parameterization of the CPU in the “Memory” tab).

It is similar with the block nesting depth. The nesting depth per priority class permissible for a CPU is the total of the nesting depth of the “normal” processing and the nesting depth of the synchronous error processing.

In the case of S7-400, another synchronous error OB can be called in an error OB. The

Table 23.2 SFC Parameters for Synchronous Error Handling

SFC	Parameter	Declaration	Data Type	Contents, Description
36	PRGFLT_SET_MASK	INPUT	DWORD	New (additional) programming error filter
	ACCFLT_SET_MASK	INPUT	DWORD	New (additional) access error filter
	RET_VAL	RETURN	INT	W#16#0001 = The new filter overlaps the existing filter
	PRGFLT_MASKED	OUTPUT	DWORD	Complete programming error filter
	ACCFLT_MASKED	OUTPUT	DWORD	Complete access error filter
37	PRGFLT_RESET_MASK	INPUT	DWORD	Programming error filter to be reset
	ACCFLT_RESET_MASK	INPUT	DWORD	Access error filter to be reset
	RET_VAL	RETURN	INT	W#16#0001 = The new filter contains bits that are not set (in the current filter)
	PRGFLT_MASKED	OUTPUT	DWORD	Remaining programming error filter
	ACCFLT_MASKED	OUTPUT	DWORD	Remaining access error filter
38	PRGFLT_QUERY	INPUT	DWORD	Programming error filter to be queried
	ACCFLT_QUERY	INPUT	DWORD	Access error filter to be queried
	RET_VAL	RETURN	INT	W#16#0001 = The query filter contains bits that are not set (in the current filter)
	PRGFLT_CLR	OUTPUT	DWORD	Programming error filter with error messages
	ACCFLT_CLR	OUTPUT	DWORD	Access error filter with error messages

block nesting depth for a synchronous error OB is 3 for S7-400 CPUs and 4 for S7-300 CPUs.

You can disable and enable a synchronous error OB call with system functions SFC 36 MSK_FLT, SFC 37 DMSK_FLT and SFC 38 READ_ERR.

23.2 Synchronous Error Handling

The following system functions are provided for handling synchronous errors:

- ▷ SFC 36 MSK_FLT
Mask synchronous errors
(disable OB call)
- ▷ SFC 37 DMSK_FLT
Unmask synchronous error
(re-enable OB call)
- ▷ SFC 38 READ_ERR
Read error register

The operating system enters the synchronous error in the diagnostic buffer without regard to the use of system functions SFC 36 to SFC 38.

The parameters for these system functions are listed in Table 23.2.

23.2.1 Error Filters

The error filters are used to control the system functions for synchronous error handling. In the programming error filter, one bit stands for each programming error detected; in the access error filter, one bit stands for each access error detected. When you define an error filter, you set the bit that stands for the synchronous error you want to mask, unmask or query. The error filters returned by the system functions show a “1” for synchronous errors that are still masked or which have occurred.

The access error filter is shown in Table 23.3; the Error Code column shows the contents of variable OB122_SW_FLT in the start information for OB 122.

The programming error filter is shown in Table 23.4; the Error Code column shows the contents of variable OB121_SW_FLT in the start information for OB 121.

Table 23.3 Assignment of access Error Filter

Bit	Error Code	Assignment
2	B#16#42	I/O access error when reading S7-300 and CPU 417: the module is not present or does not acknowledge S7-400 (except CPU 417): an existing module does not acknowledge an I/O access (time-out)
3	B#16#43	I/O access error when writing S7-300 and CPU 417: the module is not present or does not acknowledge S7-400 (except CPU 417): an existing module does not acknowledge an I/O access (time-out)

Table 23.4 Programming Error Filter

Bit	Error Code	Contents
1	B#16#21	BCD conversion error (pseudo-tetrad detected during conversion)
2	B#16#22	Area length error on read (address not within area limits)
3	B#16#23	Area length error on write (address not within area limits)
4	B#16#24	Area length error on read (wrong area in area pointer)
5	B#16#25	Area length error on write (wrong area in area pointer)
6	B#16#26	Invalid timer number
7	B#16#27	Invalid counter number
8	B#16#28	Address error on read (bit address ≤ 0 in conjunction with byte, word or doubleword access and indirect addressing)
9	B#16#29	Address area on write (bit address ≤ 0 in conjunction with byte, word or doubleword access and indirect addressing)
16	B#16#30	Write error, global data block (write-protected block)
17	B#16#31	Write error, instance data block (write-protected block)
18	B#16#32	Invalid number of a global data block (DB register)
19	B#16#33	Invalid number of an instance data block (DI register)
20	B#16#34	Invalid number of a function (FC)
21	B#16#35	Invalid number of a function block (FB)
26	B#16#3A	Called data block (DB) does not exist
28	B#16#3C	Called function (FC) does not exist
30	B#16#3E	Called function block (FB) does not exist

The error filter bits not listed in the tables are not relevant to the handling of synchronous errors.

23.2.2 Masking Synchronous Errors

System function **SFC 36 MSK_FLT** disables synchronous error OB calls via the error filters. A “1” in the error filters indicates the synchronous errors for which the OBs are not to be called (the synchronous errors are “masked”).

The masking of synchronous errors in the error filters is in addition to the masking stored in the operating system's memory. SFC 36 returns a function value indicating whether a (stored) masking already exists on at least one bit for the masking specified at the input parameters (W#16#0001).

SFC 36 returns a “1” in the output parameters for all currently masked errors.

If a masked synchronous error event occurs, the respective OB is not called and the error is

Table 23.5 Parameters for SFC 44 REPL_VAL

SFC	Parameter Name	Declaration	Data Type	Contents, Description
44	VAL	INPUT	DWORD	Substitute value
	RET_VAL	RETURN	INT	Error information

entered in the error register. The Disable applies to the current priority class (priority level). For example, if you were to disable a synchronous error OB call in the main program, the synchronous error OB would still be called if the error were to occur in an interrupt service routine.

23.2.3 Unmasking Synchronous Errors

System function **SFC 37 DMSK_FLT** enables the synchronous error OB calls via the error filters. You enter a “1” in the filters to indicate the synchronous errors for which the OBs are once again to be called (the synchronous errors are “unmasked”). The entries corresponding to the specified bits are deleted in the error register. SFC 37 returns W#16#0001 as function value if no (stored) masking already exists on at least one bit for the unmasking specified at the input parameters.

SFC 37 returns a “1” in the output parameters for all currently masked errors.

If an unmasked synchronous error occurs, the respective OB is called and the event entered in the error register. The Enable applies to the current priority class (priority level).

23.2.4 Reading the Error Register

System function **SFC 38 READ_ERR** reads the error register. You must enter a “1” in the error filters to indicate the synchronous errors whose entries you want to read. SFC 38 returns W#16#0001 as function value when the selection specified in the input parameters included at least one bit for which no (stored) masking exists.

SFC 38 returns a “1” in the output parameters for the selected errors when these errors occurred, and deletes these errors in the error register when they are queried. The synchro-

nous errors that are reported are those in the current priority class (priority level).

23.2.5 Entering a Substitute Value

SFC 44 REPL_VAL allows you to enter a substitute value in accumulator 1 from within a synchronous error OB. Use SFC 44 when you can no longer read any values from a module (for instance when a module is defective). When you program SFC 44, OB 122 (“access error”) is called every time an attempt is made to access the module in question. When you call SFC 44, you can load a substitute value into the accumulator; the program scan is then resumed with the substitute value. Table 23.5 lists the parameters for SFC 44.

You may call SFC 44 in only one synchronous error OB (OB 121 or OB 122).

23.3 Asynchronous Errors

Asynchronous errors are errors which can occur independently of the program scan. When an asynchronous error occurs, the operating system calls one of the organization blocks listed below:

- OB 80 Timing error
- OB 81 Power supply error
- OB 82 Diagnostic interrupt
- OB 83 Insert/remove module interrupt
- OB 84 CPU hardware fault
- OB 85 Program execution error
- OB 86 Rack failure
- OB 87 Communication error
- OB 88 Processing abort

The OB 82 call (diagnostic interrupt) is described in detail in Chapter 23.4, “System Diagnostics”.

On the S7-400H, there are three additional asynchronous error OBs:

OB 70 I/O redundancy errors

OB 72 CPU redundancy errors

OB 73 Communications redundancy errors

The call of these asynchronous error organization blocks can be disabled and enabled with system functions SFC 39 DIS_IRT and SFC 40 EN_IRT, and delayed and enabled with system functions SFC 41 DIS_AIRT and SFC 42 EN_AIRT.

Timing errors OB 80

The operating system calls organization block OB 80 when one of the following errors occurs:

- ▷ Cycle monitoring time exceeded
- ▷ OB request error (the requested OB is still executing or an OB was requested too frequently within a given priority class)
- ▷ Time-of-day interrupt error (TOD interrupt time past because clock was set forward or after transition to RUN)

If no OB 80 is available and a timing error occurs, the CPU goes to STOP. The CPU also goes to STOP if the OB is called a second time in the same program scan cycle due to a cycle time violation.

Power supply errors OB 81

The operating system calls organization block OB 81 if one of the following errors occurs:

- ▷ At least one backup battery in the central controller or in an expansion unit is empty
- ▷ No battery voltage in the central controller or in an expansion unit
- ▷ 24 V supply failed in central controller or in an expansion unit

OB 81 is called for incoming and outgoing events. If there is no OB 81, the CPU continues functioning when a power supply error occurs.

Insert/remove module interrupt OB 83

The operating system monitors the module configuration once per second. An entry is made in the diagnostic buffer and in the system status list each time a module is inserted or removed in RUN, STOP or START-UP mode.

In addition, the operating system calls organization block OB 83 if the CPU is in RUN mode. If there is no OB 83, the CPU goes to STOP on an insert/remove module interrupt.

As much as a second can pass before the insert/remove module interrupt is generated. As a result, it is possible that an access error or an error relating to the updating of the process image could be reported in the interim between removal of a module and generation of the interrupt.

If a suitable module is inserted into a configured slot, the CPU automatically parameterizes that module, using data records already stored on that CPU. Only then is OB 83 called in order to signal that the connected module is ready for operation.

CPU hardware faults OB 84

The operating system calls organization block OB 84 when an interface error (MPI network, PROFIBUS DP) occurs or disappears. If there is no OB 84, CPUs with older operating systems go to STOP on a CPU hardware fault.

Program execution errors OB 85

The operating system calls organization block OB 85 when one of the following errors occurs:

- ▷ Start request for an organization block which has not been loaded
- ▷ Error occurred while the operating system was accessing a block (for instance no instance data block when a system function block (SFB) was called)
- ▷ I/O access error while executing (automatic) updating of the process image on the system side

On the S7-400 CPUs, OB 85 is called at an I/O access error on the system side, i.e. when updating the process image in each cycle. The substitute value or zero is then entered in the relevant

byte in the process-image input table at every update.

On the S7-300 CPUs, OB 85 is not called in the event of an I/O access error during automatic updating of the process image. At the first errored access, the substitute value or zero is entered in the relevant byte; it is then no longer updated.

With appropriately equipped CPUs, you can use CPU parameterization to influence the call mode of OB 85 in the event of an I/O access error on the system side:

- ▷ OB 85 is called every time. The affected input byte is overwritten with the substitute value or with zero each time.
- ▷ OB 85 is called in the event of the first error with the attribute “incoming”. An affected input byte is only overwritten with the substitute value or with zero the first time; following this it is no longer updated. If the error is then corrected, OB 85 is called with the attribute “outgoing”; following this, it is updated “normally”.
- ▷ OB 85 is not called in the event of an access error. Affected input bytes are overwritten once with the substitute value or zero, and then no longer updated.

If there is no OB 85, the CPU goes to STOP on a program execution error.

Rack failure OB 86

The operating system calls organization block OB 86 if it detects the failure of a rack (power failure, line break, defective IM; not with S7-300), a DP master system or a distributed I/O station (PROFIBUS DP or PROFINET IO). OB 86 is called for both incoming and leaving errors.

In multicomputing, OB 86 is called in all CPUs if a rack fails.

If there is no OB 86, the CPU goes to STOP if a rack failure occurs.

Communication error OB 87

The operating system calls organization block OB 87 when a communication error occurs. Some examples of communication errors are:

- ▷ Invalid frame identification or frame length detected during global data communication
- ▷ Sending of diagnostic entries not possible
- ▷ Clock synchronization error
- ▷ GD status cannot be entered in a data block

If there is no OB 87, the CPU goes to STOP when a communication error occurs.

Processing abort OB 88

The operating system calls organization block OB 88 when processing of a block in the user program is aborted. Possible reasons for an abort include:

- ▷ With a synchronous error, the permissible block nesting depth is exceeded.
- ▷ With a block call, the permissible nesting depth is exceeded.
- ▷ A fault has occurred when allocating the local data of a block.

If there is no OB 88, the CPU goes to STOP if a processing abort occurs. The CPU also goes to STOP if the OB is called in priority class 28.

I/O redundancy error OB 70

The operating system of an H CPU calls organization block OB 70 if a redundancy loss occurs on PROFIBUS DP, e.g. in the event of a bus failure on the active DP master or in the event of a fault in the interface of a DP slave.

If OB 70 does not exist, the CPU continues to operate in the event of an I/O redundancy error.

CPU redundancy error OB 72

The operating system of an H CPU calls organization block OB 72 if one of the following events occurs:

- ▷ Redundancy loss of the CPU
- ▷ Comparison error (e.g. in RAM, in the PIQ)
- ▷ Standby-master changeover
- ▷ Synchronization error
- ▷ Error in a SYNC submodule
- ▷ Update abort

If OB 72 does not exist, the CPU continues to operate in the event of a CPU redundancy error.

Communications redundancy error OB 73

The operating system of a fault-tolerant CPU calls the organization block OB 73 when the redundancy of a fault-tolerant S7 connection is lost for the first time. If at least one fault-tolerant S7 connection signals a loss of redundancy, the OB 73 is not called again when there is a further loss of redundancy.

If OB 73 does not exist, the CPU continues to operate in the event of a communications redundancy error.

- ▷ The diagnostic event is passed on to the CPU's operating system
- ▷ A diagnostic interrupt is generated if you have set the parameters enabling such interrupts (by default, diagnostic interrupts are disabled).

All diagnostic events reported to the CPU operating system are entered in a *diagnostic buffer* in the order in which they occurred, and with date and time stamp. The diagnostic buffer is a battery-backed memory area on the CPU which retains its contents even in the event of a memory reset. The diagnostic buffer is a ring buffer whose size depends on the CPU. When the diagnostic buffer is full, the oldest entry is overwritten by the newest.

You can read out the diagnostic buffer with a programming device at any time. In the CPU's *System Diagnostics* parameter block you can specify whether you want expanded diagnostic entries (all OB calls). You may also specify whether the last diagnostic entry made before the CPU goes to STOP should be sent to a specific node on the MPI bus.

23.4 System Diagnostics**23.4.1 Diagnostic Events and Diagnostic Buffer**

System diagnostics is the detection, evaluation and reporting of errors occurring in programmable controllers. Examples are errors in the user program, module failures or wirebreaks on signaling modules. These *diagnostic events* may be:

- ▷ Diagnostic interrupts from modules with this capability
- ▷ System errors and CPU mode transitions
- ▷ User messages via system functions.

Modules with diagnostic capabilities distinguish between programmable and non-programmable diagnostic events. Programmable diagnostic events are reported only when you have set the parameters necessary to enable diagnostics. Non-programmable diagnostic events are always reported, regardless of whether or not diagnostics have been enabled. In the event of a reportable diagnostic event,

- ▷ The fault LED on the CPU goes on

23.4.2 Writing User Entries in the Diagnostic Buffer

System function **SFC 52 WR_USMSG** writes an entry in the diagnostic buffer which may be sent to all nodes on the MPI bus Table 23.6 lists the parameters for SFC 52.

The entry in the diagnostic buffer corresponds in format to that of a system event, for instance the start information for an organization block. Within the permissible boundaries, you may choose your own event ID (EVENTN parameter) and additional information (INFO1 and INFO2 parameters).

Table 23.6 Parameters for SFC 52 WR_USMSG

SFC	Parameter Name	Declaration	Data Type	Contents, Description
52	SEND	INPUT	BOOL	If "1": Send is enabling
	EVENTN	INPUT	WORD	Event ID
	INFO1	INPUT	ANY	Additional information 1 (one word)
	INFO2	INPUT	ANY	Additional information 2 (one doubleword)
	RET_VAL	RETURN	INT	Error information

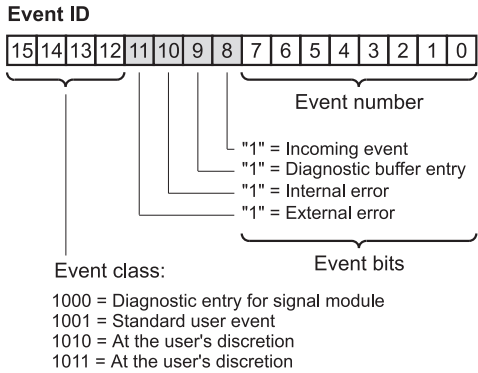


Figure 23.1 Event ID for Diagnostic Buffer Entries

The event ID is identical to the first two bytes of the buffer entry (Figure 23.1). Permissible for a user entry are the event classes 8 (diagnostic entries for signal modules), 9 (standard user events), A and B (arbitrary user events).

Additional information (INFO1) corresponds to bytes 7 and 8 of the buffer entry (one word) and additional information 2 (INFO2) to bytes 9 to 12 (one doubleword). The contents of both variables may be of the user's own choice.

Set SEND to "1" to send the diagnostic entry to the relevant node. Even if sending is not possible (because no node is logged in or because the Send buffer is full, for example), the entry is still made in the diagnostic buffer (when bit 9 of the event ID is set).

23.4.3 Evaluating Diagnostic Interrupts

When a diagnostic interrupt is incoming or outgoing, the operating system interrupts scanning of the user program and calls **organization block OB 82**. If OB 82 has not been programmed, the CPU goes to STOP on a diagnostic interrupt. You can disable or enable the processing of OB 82 with the system functions SFC 39 DIS_IRT and SFC 40 EN_IRT, and delay or enable it with the system functions SFC 41 DIS_AIRT and SFC 42 EN_AIRT

PROFIBUS DPV1 slaves can also generate a diagnostic interrupt if the master CPU is at STOP. A diagnostic interrupt triggered with the CPU at STOP is acknowledged, but not pro-

cessed. Calling the organization block OB 82 is not carried out subsequently when the CPU goes to RUN.

Table 23.7 shows the start information of the diagnostic interrupt OB 82. In the first byte of the start information, B#16#39 stands for an incoming diagnostic interrupt and B#16#38 for a leaving diagnostic interrupt. The sixth byte gives the address identifier (B#16#54 stands for an input, B#16#55 for an output); the subsequent INT variable contains the address of the module that generated the diagnostic interrupt. The next four bytes contain the diagnostic information provided by that module.

You can use system function SFC 59 RD_REC (read data record) in OB 82 to obtain detailed error information. The diagnostic information are consistent until OB 82 is exited, that is, they remain "frozen". Exiting of OB 82 acknowledges the diagnostic interrupt on the module.

A module's diagnostic data are in data records DS 0 and DS 1. Data record DS 0 contains four bytes of diagnostic data describing the current status of the module. The contents of these four bytes are identical to the contents of bytes 8 to 11 of the OB 82 start information. Data record DS 1 contains the four bytes from data record DS 0 and, in addition, the module-specific diagnostic data.

When using a CPU with DPV1 capability and a corresponding slave, you can use the system function block SFB 54 RALRM to obtain further information on the diagnostic interrupt.

23.4.4 Reading the System Status List

The system status list (SSL) describes the current status of the programmable controller. Using information functions, the list can be read but not modified. Since the complete system status list is extremely large, reading is carried out in sublists and sublist extracts. Sublists are virtual lists, which means that they are made available by the CPU operating system only on request.

The SSL ID exists in order to identify a sublist. This contains the module type class to which the list applies, the number of the sublist extract, and the actual SSL sublist number (Figure 23.2). You are provided with the desired

Table 23.7 Start Information of Organization Block OB 82 (Diagnostics Interrupt)

Byte	Variable Name	Data Type	Contents, Description
0	OB82_EV_CLASS	BYTE	B#16#38 = DOWN event B#16#39 = UP event
1	OB82_FLT_ID	BYTE	Error code (B#16#42)
2	OB82_PRIORITY	BYTE	Priority class for the diagnostic interrupt OB
3	OB82_OB_NUMBR	BYTE	OB number (B#16#52)
4	OB82_RESERVED_1	BYTE	Reserved
5	OB82_IO_FLAG	BYTE	I/O identification (B#16#54 = input, B#16#55 = output)
6...7	OB82_MDL_ADDR	WORD	Module starting address of module sending the interrupt
8.0	OB82_MDL_DEFECT	BOOL	Module fault
8.1	OB82_INT_FAULT	BOOL	Internal fault
8.2	OB82_EXT_FAULT	BOOL	External fault
8.3	OB82_PNT_INFO	BOOL	Channel fault present
8.4	OB82_EXT-VOLTAGE	BOOL	External power supply missing
8.5	OB82_FLD_CONNCTR	BOOL	Front connector missing
8.6	OB82_NO_CONFIG	BOOL	Module not parameterized
8.7	OB82_CONFIG_ERR	BOOL	Incorrect parameters in module
9	OB82_MDL_TYPE	BYTE	Bits 0 to 3: module class Bit 4: channel information present Bit 5: user information present Bit 6: diagnostics interrupt from proxy Bit 7: reserved
10.0	OB82_SUB_MDL_ERR	BOOL	User module wrong or missing
10.1	OB82_COMM_FAULT	BOOL	Communications fault
10.2	OB82_MDL_STOP	BOOL	Operating status (“0” = RUN, “1” = STOP)
10.3	OB82_WTCH_DOG_FLT	BOOL	Timeout has triggered
10.4	OB82_INT_PS_FLT	BOOL	Module-internal power supply failed
10.5	OB82_PRIM_BATT_FLT	BOOL	Flat battery
10.6	OB82_BCKUP_BATT_FLT	BOOL	Complete battery backup failed
10.7	OB82_RESERVED_2	BOOL	Reserved
11.0	OB82_RACK_FLT	BOOL	Expansion unit failed
11.1	OB82_PROC_FLT	BOOL	Processor failure
11.2	OB82_EPROM_FLT	BOOL	EPROM fault
11.3	OB82_RAM_FLT	BOOL	RAM fault
11.4	OB82_ADU_FLT	BOOL	ADC/DAC fault
11.5	OB82_FUSE_FLT	BOOL	Blown fuse
11.6	OB82_HW_INTR_FLT	BOOL	Hardware interrupt lost
11.7	OB82_RESERVED_3	BOOL	Reserved
12...19	OB82_DATE_TIME	DT	Acquisition time of diagnostics event

SSL ID

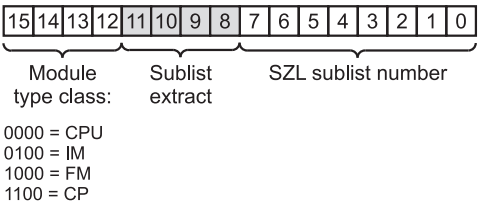


Figure 23.2 Structure of SSL ID

information together with the index which specifies an object of a sublist. As standard, the CPU provides information on the automation system, but FM and CP modules can also use this service in order to make information available (see module documentation). The possible system status lists for a CPU are described in the operations description.

Reading the header information

Use the SSL ID W#16#0Fxx to read the header information of an SSL sublist without the associated data record (xx = SSL sublist number). The SSL_HEADER. N_DR parameter (number of data records) then returns the maximum possible number of data records of the sublist extract which the module can deliver with an SSL job. With dynamic sublists, the value can be greater than the actual number which can be read. The length of a data record is specified in SSL_HEADER. With this data in the header information, it is possible e.g. to create a sufficiently large data buffer for the associated SSL sublist in the startup.

SFC 51 RDSYSST
Read SSL sublist

Use system function SFC 51 RDSYSST to read a sublist or a sublist extract of the system status list (SSL). The parameters for SFC 51 are listed in Table 23.8.

REQ = “1” initiates the read operation, and BUSY = “0” tells you when it has been completed. The operating system can execute several asynchronous read operations quasi simultaneously; how many depends on the CPU being used. If SFC 51 reports a lack of resources via the function value (W#16#8085), you must resubmit your read request.

The contents of parameters SSL_ID and INDEX are CPU-dependent. If the INDEX parameter is not required to provide information, its assignment is irrelevant. The SSL_HEADER parameter is of data type STRUCT, with variables LENGTHDR (data type WORD) and N_DR (WORD) as components. LENGTHDR contains the length of a data record, N_DR the number of data records read.

Use the DR parameter to specify the variable or data area in which SFC 51 is to enter the data records. For example, P#DB200.DBX0.0 WORD 256 would provide an area of 256 data words in data block DB 200, beginning with DBB 0. If the area provided is of insufficient capacity, as many data records as possible will be entered. Only complete data records are transferred. The specified area must be able to accommodate at least one data record.

Table 23.8 Parameters for SFC 51 RDSYSST

SFC	Parameter	Declaration	Data Type	Contents, Description
51	REQ	INPUT	BOOL	If “1”: Submit request
	SSL_ID	INPUT	WORD	Sublist ID
	INDEX	INPUT	WORD	Type or number of the sublist object
	RET_VAL	RETURN	INT	Error information
	BUSY	OUTPUT	BOOL	If “1”: Read not yet completed
	SSL_HEADER	OUTPUT	STRUCT	Length and number of data records read
	DR	OUTPUT	ANY	Field for data records read

23.5 Web Server

CPUs with an Ethernet interface can have a Web server that provides information from the CPU. To read out the information via the company's intranet or the Internet, you require a Web browser, such as Internet Explorer, Version 6.0 or higher, that displays the information on HTML pages.

23.5.1 Activating the Web server

You can activate the Web server when parameterizing the CPU with the Hardware Configuration. When a CPU is selected, EDIT → OBJECT PROPERTIES selects the “Web” tab on which you activate the Web server and select the language for the message texts and the entries in the diagnostics buffer. It is also possible to select several languages, depending on the memory capacity of the CPU.

The languages installed with STEP 7 are available. You can establish agreement with the languages installed in the project in the SIMATIC Manager: Select OPTIONS → LANGUAGE FOR DISPLAY DEVICES and define the languages. You can define the access privileges to the Web server site by means of a user list. The Web server is ready for use after the configuration data has been loaded onto the CPU.

23.5.2 Reading out Web information

To dial the CPU in the Web browser, enter the IP address of the CPU in the “Address” field in the form *http://aaa.bbb.ccc.ddd*. You can find the IP address of the CPU in the Object Properties of the PROFINET interface on the “General” tab.

The Web server also supports the Terminal Service of Windows so that thin client solutions can be used with mobile devices or HMI stations with the thin client option under Windows CE. In this case, specify the address in the form *http://aaa.bbb.ccc.ddd/basic*.

You can navigate to further information from the start page of the CPU. Please note that the information offered is static and you have to provide for updating of the screen content yourself. The current information from the CPU is

used for printing out, regardless of what is on the display.

Note: Protect your Web server against unauthorized access by means of a firewall.

23.5.3 Web information

The Web server can provide the following information in a suitably equipped CPU:

- ▷ Start page with general CPU information
- ▷ Identification data
- ▷ Diagnostics buffer
- ▷ Messages (without acknowledge facility)
- ▷ PROFINET interface
- ▷ Variable status
- ▷ Variable tables

The first page provided by the Web server is the Welcome page. From here, click on ENTER to reach the Start page. If you want to skip this intro page in future, activate the option “Skip Intro”.

Start and identification

The Start page shows you general information and the status of the CPU at the time of the query. The Identification page contains the key data of the CPU, such as the plant designation, location designation, and order number.

Diagnostics buffer

You see the contents of the diagnostics buffer on this page. Select the number of diagnostics buffer entries per display interval. Detailed information about the selected event is displayed.

You can select the display language in the top right window. If the selected language is not configured, the information is displayed in hexadecimal code.

Messages

Messages are displayed in chronological order, including the date and time. You cannot acknowledge the messages via the Web browser.

You can search for specific information with filter settings. With sort functions, you can sort

the messages according to message number or status. Detailed information about the selected message is displayed.

You can select the display language in the top right window. If the selected language is not configured, the information is displayed in hexadecimal code.

PROFINET interface

Information on the PROFINET interface is found on the pages “Parameters” and “Statistics”. The MAC address and the IP address are displayed, for example, as are statistical analyses of sent and received data packages.

Variable status

You can monitor the status of up to 50 variables on this page. When you specify the address of the variables and the display format, you receive the value of the variable.

You can select the display language in the window at the top right. When specifying addresses, please note that the mnemonics for English (e.g. I for input) differs from those of other languages (E for “Eingang” in German, for example). Syntax errors are indicated in red.

Variable tables

The Web server allows you to monitor up to 50 variable tables with up to 200 variables each. The memory space available in the CPU might not be sufficient to make use of all the possibilities. If variable tables are displayed incomplete, reduce the memory required by the mes-

sages and symbol comments, if possible, use only one language, and keep the number of variables per table low.

Select one of the configured variable tables for display. First, you must prepare the variable table for use by the Web server. If you select EDIT → OBJECT PROPERTIES when a variable table is selected, or if you create a new variable table, the Properties window opens. Enter *VAT-toWEB* as the family on the “General – Part 2” tab, or alternatively, activate the “Web server” checkbox.

SFC 99 WWW
Synchronize user websites

Using the configuration tool S7-Web2PLC, you can integrate self-generated websites into the CPU's Web server. The websites can show CPU data, controlled either by direct access or by the user program.

These websites are saved in special data blocks – the “Fragment DBs”. One data block – the “Web control DB” – contains the structure information required to edit the websites.

The system function SFC 99 WWW makes the user websites known to the CPU's operating system. It must be called once for this, e.g. during startup.

In addition, the SFC 99 synchronizes the user program and the Web server. It must be called cyclically for this purpose, e.g. in the main program.

Table 23.9 shows the parameters of the SFC 99 WWW.

Tabelle 23.9 Parameter der SFC 99 WWW

SFC	Parametername	Deklaration	Datentyp	Belegung, Beschreibung
99	CTRL_DB	INPUT	BLOCK_DB	Web-Control-DB
	RET_VAL	RETURN	INT	Fehlerinformation

Appendix

This section of the book contains useful supplements to the LAD and FBD programming languages, an overview of the contents of the STEP 7 Block Libraries, and a function overview of all LAD and FBD elements.

- ▷ You can also provide blocks with LAD/FBD program with **block protection**. For this purpose, you use the source-oriented Editor in the STL programming language.
- ▷ You can use a further function of STL, **indirect addressing**, to transfer data areas in the LAD and FBD programming languages; the addresses of these data areas are then not calculated until runtime. The “LAD_Book” and “FBD_Book” libraries which you can download from the publisher's Website (see page 8) each contain a “Sample Message Frame” showing how to set up and transfer data areas.
- ▷ The standard STEP 7 package includes **block libraries** with loadable functions and function blocks and with block headers and interface descriptions for system blocks (SFCs and SFBs).
- ▷ A **function overview** of all LAD and FBD functions completes the book.

You can download the archive libraries “LAD_Book” and “FBD_Book” from the publisher's Website (see page 8). You retrieve these libraries under the SIMATIC Manager with FILE → RETRIEVE. Select the archive from the dialog field displayed. You define the target directory in the next dialog box. In general, libraries are located under ...\\STEP7\\S7LIBS, but you can choose any other directory, for example ...\\STEP7\\S7PROJ, which normally contains the projects.

The “LAD_Book” and “FBD_BOOK” libraries each contain eight programs which are essentially illustrative examples of LAD and FBD representations. Two extensive examples show the programming of functions, function blocks and local instances (Conveyor Example) and the handling of data (Message Frame Example). The memory requirements are approximately 2 MB.

To try out an example, set up a project that corresponds to your hardware configuration and copy the program, including the symbol table, from the library to the project. Now you can test the example online.

- 24 Supplements to Graphic Programming**
Block Protection; Indirect Addressing; Message Frame Example
- 25 Block Libraries**
Organization Blocks, System Function Blocks, IEC Function Blocks, S5-S7 Converting Blocks, TI-S7 Converting Blocks, PID Control Blocks, Communication Blocks
- 26 LAD Function Overview**
All LAD functions
- 27 FBD Function Overview**
All FBD functions

24 Supplements to Graphic Programming

24.1 Block Protection

The keyword `KNOW_HOW_PROTECT` represents block protection. You cannot view, print or modify a block with this attribute. The Editor only displays the block header and the declaration table with the block parameters. In source-oriented input, you can protect any block yourself with `KNOW_HOW_PROTECT`. This means that no one, including yourself, can view the compiled block (keep the source file in a safe place!).

You can enter the `KNOW_HOW_PROTECT` block protection with `STL`, and it must be source-oriented. To do so, proceed as follows:

- 1) Create the block under LAD or FBD in the usual way. Later, this block will be overwritten in the user program *Blocks* by the block with the keyword. If you want to retain the (original) block (strongly recommended when entering block protection), you can store the block in, for example, a (user-created) library before entering the keyword. You can also store your entire user program in this way.
- 2) Create a source container. If there is no object *Sources* under the *S7 program* (on the same level as the user program *Blocks*), you must create one: Select the *S7 program* and insert the object *Sources* with `INSERT → S7 SOFTWARE → SOURCE FOLDER`.
- 3) Generate an STL source from the block. Change to the Editor (via the taskbar, for example, or open any block in *Blocks* and then close it again) and select the menu item `FILE → GENERATE SOURCE`. In the dialog form displayed, set your project, select the object *Sources*, and assign a name for the source file under "Object Name". Confirm with "OK". The next dialog form shows you all blocks in the

Blocks container; select the block(s) from which you want to create a source file. Confirm with "OK".

- 4) Open the source file (for example by double-clicking on the source file symbol in the SIMATIC Manager or with the Editor and `FILE → OPEN`). You now see the ASCII source of your LAD/FBD block. If you have previously selected several blocks, these blocks will be arranged in order in the source file.

The entries for a code block are in the following order:

- ▷ Keyword for the block type (`FUNCTION`, `FUNCTION_BLOCK`, `ORGANIZATION_BLOCK`) with specification of the address. This may be followed by the block title (starting with `TITLE=...`) and the block comment (starting with `//...`).
- ▷ Block attributes (depending on whether you have filled in fields in the Properties page for the block and, if so, how many).
- ▷ Variable declaration (several sections with the keywords `VAR xxx, ..., END_VAR`); depending on whether you have declared local block variables and, if so, which ones.
- ▷ Program, starts with `BEGIN` and ends with the keyword for the end of the block (for example `END_FUNCTION_BLOCK`).

The entries for a data block are in the following order:

- ▷ Keyword for the block type (`DATA_BLOCK`) with specification of the address.
- ▷ Block attributes (depending on whether you have filled in fields in the Properties page for the block and, if so, how many).
- ▷ Variable declaration (starting with `STRUCT` and ending with `END_STRUCT`) or, in the case of instance data blocks, the address of the associated function block.

- ▷ Variable initialization, starting with BEGIN and ending with END_DATA_BLOCK.
- 5) You enter the keyword KNOW_HOW_PROTECT in the source file, in its own line in each case, following the block attributes and before the variable declaration. If you have created a source from several blocks, enter the keywords in all selected blocks. Finally, store the source file.
- 6) Compile the source file with FILE → COMPILE. The compiler creates a block with the specified block attributes (in the creation language STL in the case of code blocks; the creation language is not significant here since KNOW_HOW_PROTECT means the block can no longer be viewed or printed out). The compiled (new) block is in user program *Blocks* and replaces the (old) block with the same number.

You can set the program editor using OPTIONS → CUSTOMIZE on the “Sources” tab so that it automatically generates a source when saving a block. You can use the “Run” button to generate new sources from existing blocks.

24.2 Indirect Addressing

With the programming language STL, you have a method of accessing operands whose addresses are not calculated until runtime. This is also possible to a limited degree in LAD or FBD: You can wait until runtime to define which data areas you want to copy with SFC 20 BLKMOV.

However, first some useful information on pointers.

24.2.1 Pointers: General Remarks

For indirect addressing, you require a data format that contains the bit address as well as the byte address and, if applicable, the operand area. This data format is a pointer. A pointer is also used to point to an operand.

There are three types of pointers:

- ▷ Area pointers; these are 32 bits long and contain a specific operand or its address
- ▷ DB pointers; these are 48 bits long and in addition to the area pointer they also contain the number of the data block
- ▷ ANY pointers; these are 80 bits long and, in addition to the DB pointer, they contain further information such as the data type of the operand

24.2.2 Area Pointer

The area pointer contains the operand address and, if applicable, the operand area. Without operand area, it is an area-internal pointer; if the pointer also contains an operand area, it is referred to as an area-crossing pointer.

The notation for constant representation is as follows:

P#y.x for an area-internal pointer
e.g. P#22.0

P#Zy.x for an area-crossing pointer
e.g. P#M22.0

where x = bit address, y = byte address and Z = area. You specify the operand ID as the area. The contents of bit 31 differentiates the two pointer types.

Figure 24.1 shows all pointer types and their contents as provided by STEP 7.

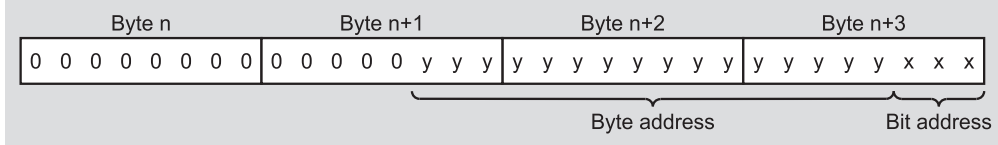
The area pointer has, in principle, a bit address that must always be specified even with digital operands; in the case of digital operands, specify 0 as the bit address. Example: You can use area pointer P#M22.0 to address memory bit M 22.0, but also memory byte MB 22, memory word MW 22 or memory doubleword MD 22.

24.2.3 DB Pointer

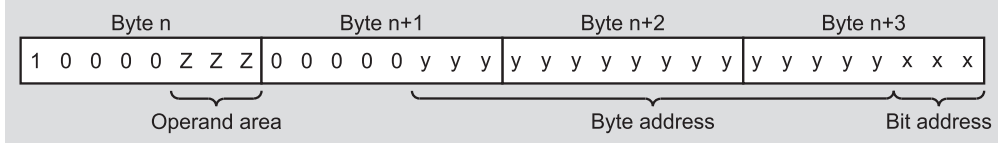
In addition to the area pointer, a DB pointer also contains a data block number in the form of a positive INT number. It specifies the data block if the area pointer points to the global data or instance data area. In all other cases, the first two bytes contain zero.

The notation of the pointer is familiar to you from full addressing of data operands. Here also, the data block and the data operand are

Area-internal pointer



Area-crossing pointer



		ANY pointer for data types	ANY pointer for timers/counters	ANY pointer for blocks
DB pointer	Byte n	16#10	16#10	16#10
	Byte n+1	Type	Type	Type
	Byte n+2	Quantity	Quantity	Quantity
	Byte n+3	Data block	16#0000	16#0000
	Byte n+4	number	Type	16#0000
	Byte n+5	Area	16#00	Number
DB pointer	Byte n	Area	Number	Number
	Byte n+1	pointer		
	Byte n+2			
	Byte n+3			
	Byte n+4			

Address area:

0	0	0	Peripheral I/O (P)
0	0	1	Inputs (I)
0	1	0	Outputs (Q)
0	1	1	Memory bits (M)
1	0	0	Global data (DBX)
1	0	1	Instance data (DIX)
1	1	0	Temporary local data (L) ¹⁾
1	1	1	Temporary local data of the predecessor block (V) ²⁾

¹⁾ Not with area-crossing addressing
²⁾ Only with block parameter transfer

Type in the ANY pointer:

Elementary data types	Complex data types
01 BOOL	0E DT
02 BYTE	13 STRING
03 CHAR	Parameter types
04 WORD	17 BLOCK_FB
05 INT	18 BLOCK_FC
06 DWORD	19 BLOCK_DB
07 DINT	1A BLOCK_SDB
08 REAL	1C COUNTER
09 DATE	1D TIMER
0A TOD	Zero pointer
0B TIME	00 NIL
0C S5TIME	

Figure 24.1 Structure of the Pointers in STEP 7

specified, separated by a period: P#DataBlock.
DataOperand

Example: P#DB 10.DBX 20.5

You can apply this pointer to a block parameter of parameter type POINTER in order to point to a data operand. The Editor uses this pointer type internally to transfer actual parameters.

24.2.4 ANY Pointer

In addition to the DB pointer, the ANY pointer also contains the data type and a repetition factor. This makes it possible to point to a data area

The ANY pointer is available in two variants: For variables with data types and for variables with parameter types. If you point to a variable with a data type, the ANY pointer contains a DB pointer, the type, and a repetition factor. If the ANY pointer points to a variable with a parameter type, it contains only the number instead of the DB pointer in addition to the type. In the case of a timer or counter function, the type is repeated in byte (n+6); byte (n+7) contains B#16#00. In all other cases, these two bytes contain the value W#16#0000.

The first byte of the ANY pointer contains the syntax ID; in STEP 7 it is always 10_{hex}. The type specifies the data type of the variables for which the ANY pointer applies. Variables of elementary data type, DT and STRING receive the type shown in Figure 24.1 and the quantity 1.

If you apply a variable of data type ARRAY or STRUCT (also UDT) at an ANY parameter, the Editor generates an ANY pointer to the field or the structure. This ANY pointer contains the ID for BYTE (02_{hex}) as the type, and the byte length of the variable as the quantity. The data type of the individual field or structure components is not significant here. Thus, an ANY pointer points with double the number of bytes to a WORD field. Exception: A pointer to a field consisting of components of data type CHAR is also created with CHAR type (03_{hex}).

You can apply an ANY pointer at a block parameter of parameter type ANY if you want to point to a variable or an operand area. The constant representation for data types is as follows:

P#[DataBlock.]Operand Type Quantity

Examples:

- ▷ P#DB 11.DBX 30.0 INT 12
Area with 12 words in DB 11 beginning DBB 30
- ▷ P#M 16.0 BYTE 8
Area with 8 bytes beginning MB 16
- ▷ P#E 18.0 WORD 1
Input word IW 18
- ▷ P#E 1.0 BOOL 1
Input I 1.0

In the case of parameter types, you write the pointer as follows: L#Number Type Quantity

Examples:

- ▷ L#10 TIMER 1
Timer function T 10
- ▷ L#2 COUNTER 1
Counter function C 2

The Editor then applies an ANY pointer that agrees in type and quantity with the specifications in the constant representation. Please note that the operand address in the ANY pointer must also be a bit address for data types.

Specification of a constant ANY pointer makes sense if you want to access a data area for which you have not declared any variables. In principle, you can also apply variables or operands at an ANY parameter. For example, the representation "P#I 1.0 BOOL 1" is identical with "I 1.0" or the corresponding symbolic address.

If you do not specify any defaults when declaring an ANY parameter at a function block, the Editor assigns 10_{hex} to the syntax ID and 00_{hex} to the remaining bytes. The Editor then represents this (empty) ANY pointer (in the data view) as follows: P#P0.0 VOID 0.

24.2.5 "Variable" ANY Pointer

When copying with SFC 20, you specify in the source and destination parameters either an absolute-addressed area (for example, P#DB127.DBX0.0 BYTE 32) or a variable. In both cases, the source area and the destination area are fixed during programming (variable indexing is not possible even for array components). The following method is available for runtime modification of a data area created at a block parameter of type ANY:

You create a variable of data type ANY in the temporary local data and use this to initialize an ANY parameter. The Program Editor then does not generate an ANY pointer (as it would if you created a different variable), but uses the ANY variable in the temporary local data as an ANY pointer to a source or destination area. The ANY variable in the temporary local data is structured in the same way as an ANY pointer; you can now modify the individual entries at runtime.

This procedure functions not only in the case of SFC 20 BLKMOV but also with block parameters of type ANY in other blocks.

The “LAD_BOOK” and “FBD_Book” libraries which you can download from the publisher's Website (see page 8), each contain a “Message Frame Example” program, which in turn both contain an example of the “variable” ANY pointer.

24.3 Brief Description of the “Message Frame Example”

This example deals primarily with how data is handled, and is broken down as follows:

- ▷ Message frame data, shows how to handle data structures
- ▷ Time-of-day check, shows how to handle system blocks and standard blocks
- ▷ Editing the message frame, shows the use of SFC 20 BLKMOV with fixed addresses
- ▷ Indirect copying of the data area, shows an “indirect copy” function using “variable” ANY pointers
- ▷ Save message frame, shows the use of “indirect copying”

Figure 24.2 shows the program and data structure for this example.

You can find this program under “Message Frame Example” in the libraries “LAD_Book” and FBD_BOOK” which you can download from the publisher's Website (see page 8).

Message frame data

The example shows how you can define frequently occurring data structures as your own data type and how you use this data type when declaring variables and parameters.

We construct a data store for incoming and outgoing message frames: A Send mailbox with the structure of a message frame, a Receive mailbox with the same structure, and a (receive) ring buffer that is to provide intermediate storage for incoming message frames. Since the data structure of the message frame occurs frequently, we will define it as a user-defined data type (UDT) frame. The *message frame* contains a frame header whose structure we also want to give a name to. The Send mailbox and the Receive mailbox are to be data blocks that each contain a variable with the structure of the *frame*. Finally, there is also a ring buffer, a data block with an array consisting of eight components that also have the same data structure as the *frame*.

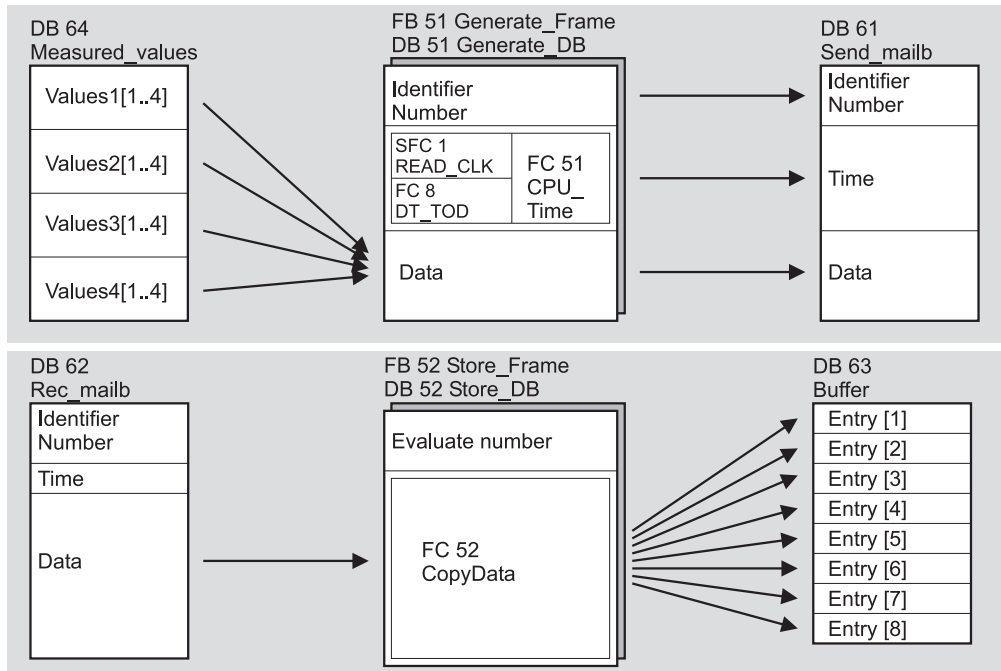
Time-of-day check

The example shows how to handle system and standard blocks (evaluating errors, copying from the library, renaming).

The time-of-day check function is to output the time-of-day in the integral CPU real-time clock as a function value. For this purpose, we require the system function SFC 1 READ_CLK, which reads the date and the time-of-day from the real-time clock in the DATE_AND_TIME or DT data format. Since we only want to read the time-of-day, we also require the IEC function FC 8 DT_TOD. This function fetches the time-of-day in the TIME_OF_DAY or TOD format from the DT data format.

Error evaluation

The system functions signal an error via binary result parameter BR and via function value RET_VAL. An error occurred if binary result BR = “0”; the function value is then also negative (bit 15 is set). The IEC standard functions signal an error only via the binary result. Both types of error evaluation are shown in the example. If an error was encountered, an invalid value is output for the time-of-day. In addition, the binary result is affected. After the

Message frame example**Figure 24.2** Data Structure for the Message Frame Data Example

time-of-day check function has been called, you can therefore also use the binary result to see if an error has occurred.

Offline programming of system functions

Before saving the input block, system function SFC 1 and standard function FC 8 must be included in the offline user program. Both functions are included in the STEP 7 standard package. You will find these functions in the block libraries provided. (For the system functions integrated into the CPU, the library contains only an interface description, not the actual system functions program. The function can be called offline via this interface description; the interface description is not transferred to the CPU. Loadable functions such as the IEC functions are available in the library as executable programs.)

Select the *Standard Library* with FILE → OPEN under the SIMATIC Manager and open the *System Function Blocks* library. Under *Blocks*, you

will find all interface descriptions for the system functions. If you still have the project window of your project open, you can display the two windows side-by-side with WINDOW → ARRANGE → VERTICALLY and “drag” the selected system functions into your program using the mouse (select the SFC with the mouse, hold the mouse key down, drag to *Blocks* or into its open window, and “drop”). Copy standard function FC 8 in the same way. You will find it on the *IEC Function Blocks* library. FC 8 is a loadable function; it therefore reserves user memory, in contrast to SFC 1.

If a standard function block is called under “Libraries” in the Program Element Catalog using the Editor, it is automatically copied into *Blocks* and entered in the Symbol Table.

Renaming standard functions

You can rename a loadable standard function. Select the standard function (for example FC 8) in the project window and click once (again) on

the designator. The name appears in a frame and you can specify a new address (for example, FC 98). If you press the F1 key while the standard function (renamed to FC 98) is still selected, you will nevertheless receive the online help for the original standard function (FC 8).

If an identically addressed block exists when copying, a dialog box appears where you can choose between overwriting and renaming.

Symbolic address

You can assign names to the system functions and the standard functions in the Symbol Table so that you can also access these functions symbolically. You can assign these names freely within the permissible limits applicable to block names. In the example, the block name in each case is selected as a symbolic name (for better identification).

Editing the message frame

The data block *Send_Mailb* is to be filled with the data for a message frame. We use a function block that has the ID and the consecutive number stored in its instance data block. The net data are stored finally in a global data block; they are copied into the Send mailbox with system function BLKMOV. We use the time-of-day check function to take the time-of-day from the CPU's real-time clock.

The first network in the function block FB *Generate_Frame* transfers the ID stored in the instance data block to the frame header. The consecutive number is incremented by +1 and is also transferred to the frame header.

The second network contains the *READ_CLK* function call that takes the time-of-day from the real-time clock and enters it in the frame header in TIME_OF_DAY format.

In the subsequent networks, you will see a method of copying selected variables at runtime with the system function SFC 20 BLKMOV and without using indirect addressing. It is therefore also not necessary to know the absolute address or the structure of the variables. The principle is extremely simple: The desired copy function is selected using comparison functions. The numbers 1 to 4 are permissible as selection criteria.

FB *Generate_Frame* is programmed in such a way that it is called via a signal edge to generate a message frame.

Indirect copying of a data area

The example shows the editing and use of a "variable" ANY pointer with graphical program elements.

The *CopyData* function copies a data area whose address and length you can set as required via block parameters. The individual block parameters correspond to the individual elements of an ANY pointer (see Chapter 24.2.4, "ANY Pointer"). The specifications in the block parameters must be valid values; they are not checked (SFC 20 BLKMOV signals a copy error in its function value parameter, which is transferred to the *CopyData* function's function value parameter).

Essential elements are the two temporary variables *SoPointer* and *DesPointer*, which are of data type ANY. They contain the ANY pointers for the system function SFC 20 BLKMOV. *SoPointer* points to the source area of the data to be transferred and *DesPointer* points to the destination area. Figure 24.3 shows the structure of the *SoPointer* variable; *DesPointer* has the same structure. The individual bytes, words and doublewords of the ANY variables are accessed via their absolute addresses.

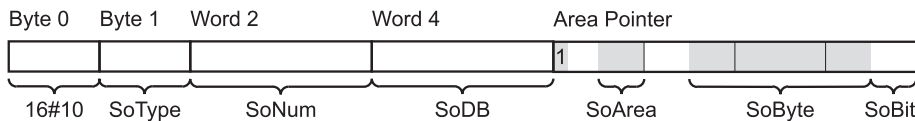


Figure 24.3 Structure of the Variable *SoPointer*

Save message frame

The example shows the use of the *CopyData* function (copying a data area with programmable address).

A message frame in the data block *Rec_Mailb* is to be written to the next location in data block *Buffer*. The block-local variable *Entry* determines the location in the ring buffer; the value of this location is used to calculate the address in the ring buffer.

The *Entry* variable has a value range of from 0 to 7. In the first network, a comparator determines whether *Entry* is less than 7. If this is the case, *Entry* is incremented by 1 in the next network, otherwise it is set to zero. *Entry* multiplied by 16 gives the absolute byte address of the next entry in the ring buffer (the data structure *Message frame* consists of 16 bytes).

The *CopyData* function, which copies the message frame from the Receive mailbox (data block DB 62) to the ring buffer (data block DB 63), is called in Network 3.

25 Block Libraries

The STEP 7 Basic software includes the **Standard Library** which contains the following library programs:

- ▷ Organization Blocks
- ▷ System Function Blocks
- ▷ IEC Function Blocks
- ▷ S5-S7 Converting Blocks
- ▷ TI-S7 Converting Blocks
- ▷ PID Control Blocks
- ▷ Communication Blocks
Communication functions
- ▷ Miscellaneous Blocks
Time synchronization and stamping

Further supplied libraries are **SIMATIC_NET_CP**, which contains the communications blocks for the CP modules in the library programs *CP 300* and *CP 400*, **Redundant IO (V1)** with blocks for module redundancy, and **Redundant IO CGP** with blocks for the redundancy of individual module channels.

You can copy blocks or interface descriptions into own projects from the library programs described.

25.1 Organization Blocks

(Prio = Default priority class)

OB Prio Designation		
1	1	Main program
10	2	Time-of-day interrupt 0
11	2	Time-of-day interrupt 1
12	2	Time-of-day interrupt 2
13	2	Time-of-day interrupt 3
14	2	Time-of-day interrupt 4
15	2	Time-of-day interrupt 5
16	2	Time-of-day interrupt 6
17	2	Time-of-day interrupt 7
OB Prio Designation		
20	3	Time-delay interrupt 0
21	4	Time-delay interrupt 1
22	5	Time-delay interrupt 2
23	6	Time-delay interrupt 3
30	7	Watchdog interrupt 0 (5 s)
31	8	Watchdog interrupt 1 (2 s)
32	9	Watchdog interrupt 2 (1 s)
33	10	Watchdog interrupt 3 (500 ms)
34	11	Watchdog interrupt 4 (200 ms)
35	12	Watchdog interrupt 5 (100 ms)
36	13	Watchdog interrupt 6 (50 ms)
37	14	Watchdog interrupt 7 (20 ms)
38	15	Watchdog interrupt 8 (10 ms)
40	16	Hardware interrupt 0
41	17	Hardware interrupt 1
42	18	Hardware interrupt 2
43	19	Hardware interrupt 3
44	20	Hardware interrupt 4
45	21	Hardware interrupt 5
46	22	Hardware interrupt 6
47	23	Hardware interrupt 7
55	2	DPV1 status interrupt
56	2	DPV1 update interrupt
57	2	DPV1 vendor interrupt
60	25	Multiprocessor interrupt

OB Prio Designation

61	25	Synchronous cycle interrupt 0
62	25	Synchronous cycle interrupt 1
63	25	Synchronous cycle interrupt 2
64	25	Synchronous cycle interrupt 3
65	25	Technology synchronous interrupt
70	25	I/O redundancy error
72	28	CPU redundancy error
73	25	Communication redundancy error
80	26	Time error ¹⁾
81	26	Power supply fault ¹⁾
82	26	Diagnostics interrupt ¹⁾
83	26	Insert/remove-module interrupt ¹⁾
84	26	CPU hardware fault ¹⁾
85	26	Priority class error ¹⁾
86	26	DP error ¹⁾
87	26	Communications error ¹⁾
88	28	Processing abort
90	29	Background processing
100	27	Warm restart
101	27	Hot restart
102	27	Cold restart
121	-	Programming error
122	-	I/O access error

¹⁾ Prio = 28 at restart

25.2 System Function Blocks**CPU clock and run-time meter**

SFC Name	Designation
0 SET_CLK	Set clock
1 READ_CLK	Read clock
2 SET_RTM	Set run-time meter
3 CTRL_RTM	Modify run-time meter
4 READ_RTM	Read run-time meter
48 SNC_RTCB	Synchronize slave clocks
64 TIME_TCK	Read system time
100 SET_CLKS	Set time and clock status
101 RTM	Use run-time meter

IEC timers and IEC counters

SFB Name	Designation
0 CTU	Up counter
1 CTD	Down counter
2 CTUD	Up/down counter
3 TP	Pulse
4 TON	On delay
5 TOF	Off delay

S7 communication

SFB Name	Designation
8 USEND	Uncoordinated send
9 URVC	Uncoordinated receive
12 BSEND	Block-oriented send
13 BRCV	Block-oriented receive
14 GET	Read data from partner
15 PUT	Write data to partner
16 PRINT	Write data to printer
19 START	Initiate cold or warm start in the partner
20 STOP	Set partner to STOP
21 RESUME	Initiate restart in the partner
22 STATUS	Check status of partner
23 USTATUS	Receive status of partner

SFC Name	Designation
62 CONTROL	Check communications status
87 C_DIAG	Determine connection status

S7 basic communication

SFC Name	Designation
65 X_SEND	Send data externally
66 X_RCV	Receive data externally
67 X_GET	Read data externally
68 X_PUT	Write data externally
69 X_ABORT	Abort external connection
72 I_GET	Read data internally
73 I_PUT	Write data internally
74 I_ABORT	Abort internal connection

Global data communications

SFC Name	Designation
60 GD_SND	Send GD packet
61 GD_RCV	Receive GD packet

S7-300C point-to-point coupling

SFB Name	Designation
60 SEND_PTP	Send data (ASCII, 3964 (R))
61 RCV_PTP	Receive data (ASCII, 3964(R))
62 RES_RCVB	Delete receive buffer (ASCII, 3964(R))
63 SEND_RK	Send data (RK 512)
64 FETCH_RK	Fetch data (RK 512)
65 SERVE_RK	Receive and provide data (RK 512)

Integral S7-300C functions

SFB Name	Designation
44 ANALOG	Positioning with analog output
46 DIGITAL	Positioning with digital output
47 COUNT	Control counter
48 FREQUENC	Control frequency measurement
49 PULSE	Control pulse-width modulation

Integrated functions CPU 312/314/614

SFB Name	Designation
29 HS_COUNT	High-speed counter
30 FREQ_MES	Frequency meter
38 HSC_A_B	Control "Counter A/B"
39 POS	Control "Positioning"
41 CONT_C	Continuous closed-loop control
42 CONT_S	Step-action control
43 PULSEGEN	Generate pulse

SFC Name	Designation
63 AB_CALL	Call assembler block

Drum

SFB Name	Designation
32 DRUM	Drum

H-CPU

SFC Name	Designation
90 H_CTRL	Control operating modes on H-CPU

Interrupt events

SFC Name	Designation
28 SET_TINT	Set time-of-day interrupt
29 CAN_TINT	Cancel time-of-day interrupt
30 ACT_TINT	Activate time-of-day interrupt
31 QRY_TINT	Query time-of-day interrupt
32 SRT_DINT	Start time-delay interrupt
33 CAN_DINT	Cancel time-delay interrupt
34 QRY_DINT	Query time-delay interrupt
35 MP_ALM	Trigger multiprocessor alarm
36 MSK_FLT	Mask synchronous errors
37 DMSK_FLT	Unmask synchronous errors
38 READ_ERR	Read event status register
39 DIS_IRT	Disable interrupt events
40 EN_IRT	Enable interrupt events
41 DIS_AIRT	Delay interrupt events
42 EN_AIRT	Enable interrupt events

Address modules

SFC Name	Designation
5 GADR_LGC	Determine logical address
49 LGC_GADR	Determine slot
50 RD_LGADR	Determine all logical addresses
70 GEO_LOG	Determine logical address
71 LOG_GEO	Determine slot

Data record transfer

SFB Name	Designation
52 RDREC	Read data record from a DP slave
53 WRREC	Write data record to a DP slave
81 RD_DPAR	Read predefined parameters

SFC Name	Designation
54 RD_DPARM	Read predefined parameter
55 WR_PARM	Write dynamic parameters
56 WR_DPARM	Write predefined parameters
57 PARM_MOD	Parameterize module
58 WR_REC	Write data record
59 RD_REC	Read data record
102 RD_DPARA	Read predefined parameters

Process image updating

SFC Name	Designation
26 UPDAT_PI	Update process-image input table
27 UPDAT_PO	Update process-image output table
79 SET	Set I/O bit field
80 RSET	Reset I/O bit field
126 SYNC_PI	Update process image partition of the inputs in isochronous mode
127 SYNC_PO	Update process image partition of the outputs in isochronous mode

Diagnostics

SFC Name	Designation
6 RD_SINFO	Read start information
51 RDSYSST	Read SYS ST sublist
52 WR_USMSG	Entry in the diagnostics buffer
99 WWW	Synchronize websites

Distributed I/O

SFB Name	Designation
54 RALRM	Receive alarm
73 RCVREC	Receive data record
74 PRVREC	Provide data record
75 SALRM	Trigger interrupt
104 IP_CONF	Set IP config.

SFC Name	Designation
7 DP_PRAL	Initiate hardware interrupt
11 DPSYC_FR	SYNC/FREEZE
12 D_ACT_DP	Deactivate or activate DP slave
13 DPNRM_DG	Read diagnostics data
14 DPRD_DAT	Read slave data
15 DPWR_DAT	Write slave data
103 DP_TOPOL	Determine bus topology

Create block-related messages

SFB Name	Designation
31 NOTIFY_8P	Messages without acknowledgment display
33 ALARM	Messages with acknowledgment display
34 ALARM_8	Messages without accompanying values
35 ALARM_8P	Messages with associated values
36 NOTIFY	Messages without acknowledgment display
37 AR_SEND	Send archive data

SFC Name	Designation
9 EN_MSG	Enable messages
10 DIS_MSG	Disable messages
17 ALARM_SQ	Messages that can be acknowledged
18 ALARM_S	Messages that are always acknowledged
19 ALARM_SC	Determine acknowledgment status

105	READ_SI	Read dynamic system resources
106	DEL_SI	Enable dynamic system resources
107	ALARM_DQ	Messages that can be acknowledged
108	ALARM_D	Messages that are always acknowledged

Copy and block functions

SFC Name		Designation
20	BLKMOV	Copy data area
21	FILL	Pre-assign data area
22	CREAT_DB	Create data block in work memory
23	DEL_DB	Delete data block
24	TEST_DB	Test data block
25	COMPRESS	Compress memory
44	REPL_VAL	Enter substitute value
81	UBLKMOV	Copy data area without gaps
82	CREA_DBL	Generate data block in load memory
83	READ_DBL	Read load memory
84	WRIT_DBL	Write load memory
85	CREA_DB	Create data block in work memory

Program control

SFC Name		Designation
43	RE_TRIGR	Retrigger cycle time monitor
46	STP	Change to STOP state
47	WAIT	Wait for delay time
78	OB_RT	Determine OB runtime
104	CIR	Configuration in RUN
109	PROTECT	Change protection level

Blocks for PROFINET CbA

SFC Name		Designation
112	PN_IN	Update inputs
113	PN_OUT	Update outputs
114	PN_DP	Update DP connections

25.3 IEC Function Blocks

String functions

FC	Name	Designation
21	LEN	Length of a STRING
20	LEFT	Left section of a STRING
32	RIGHT	Right section of a STRING
26	MID	Middle section of a STRING
2	CONCAT	Concatenate STRINGS
17	INSERT	Insert STRING
4	DELETE	Delete STRING
31	REPLACE	Replace STRING
11	FIND	Find STRING
16	I_STRNG	Convert INT to STRING
5	DI_STRNG	Convert DINT to STRING
30	R_STRNG	Convert REAL to STRING
38	STRNG_I	Convert STRING to INT
37	STRNG_DI	Convert STRING to DINT
39	STRNG_R	Convert STRING to REAL

Date and time functions

FC	Name	Designation
3	D_TOD_DT	Combine DATE and TOD to DT
6	DT_DATE	Extract DATE from DT
7	DT_DAY	Extract day-of-the-week from DT
8	DT_TOD	Extract TOD from DT
33	S5TI_TIM	Convert S5TIME to TIME
40	TIM_S5TI	Convert TIME to S5TIME
1	AD_DT_TM	Add TIME to DT
35	SB_DT_TM	Subtract TIME from DT
34	SB_DT_DT	Subtract DT from DT

Comparisons

FC	Name	Designation
9	EQ_DT	Compare DT for equal to
28	NE_DT	Compare DT for not equal to
14	GT_DT	Compare DT for greater than
12	GE_DT	Compare DT for greater than or equal to
23	LT_DT	Compare DT for less than
18	LE_DT	Compare DT for less than or equal to
10	EQ_STRNG	Compare STRING for equal to
29	NE_STRNG	Compare STRING for not equal to
15	GT_STRNG	Compare STRING for greater than
13	GE_STRNG	Compare STRING for greater than or equal to
24	LT_STRNG	Compare STRING for less than
19	LE_STRNG	Compare STRING for less than or equal to

Math functions

FC	Name	Designation
22	LIMIT	Limiter
25	MAX	Maximum selection
27	MIN	Minimum selection
36	SEL	Binary selection

25.4 S5-S7 Converting Blocks**Floating-point arithmetic**

61	GP_FGPP	Convert fixed point to floating point
62	GP_GFPF	Convert floating point to fixed point
63	GP_ADD	Add floating-point numbers
64	GP_SUB	Subtract floating-point numbers

65	GP_MUL	Multiply floating-point numbers
66	GP_DIV	Divide floating-point numbers
67	GP_VGL	Compare floating-point numbers
68	GP_RAD	Find the square root of a floating-point number

Signal functions

FC	Name	Designation
69	MLD_TG	Clock generator
70	MLD_TGZ	Clock generator with timer function
71	MLD_EZW	Initial value single blinking word-oriented
72	MLD_EDW	Initial value double blinking word-oriented
73	MLD_SAMW	Group signal word-oriented
74	MLD_SAM	Group signal
75	MLD_EZ	Initial value single blinking
76	MLD_ED	Initial value double blinking
77	MLD_EZWK	Initial value single blinking (word-oriented) bit memory
78	MLD_EZDK	Initial value double blinking (word-oriented) bit memory
79	MLD_EZK	Initial value single blinking bit memory
80	MLD_EDK	Initial value double blinking memory bit

Integrated functions

FC	Name	Designation
81	COD_B4	BCD-binary conversion 4 decades
82	COD_16	Binary-BCD conversion 4 decades
83	MUL_16	16-bit fixed-point multiplier
84	DIV_16	16-bit fixed-point divider

Basic functions

FC	Name	Designation
85	ADD_32	32-bit fixed-point adder
86	SUB_32	32-bit fixed-point subtractor
87	MUL_32	32-bit fixed-point multiplier
88	DIV_32	32-bit fixed-point divider
89	RAD_16	16-bit fixed-point square root extractor
90	REG_SCHB	Bitwise shift register
91	REG_SCHW	Word-oriented shift register
92	REG_FIFO	Buffer (FIFO)
93	REG_LIFO	Stack (LIFO)
94	DB_COPY1	Copy data area (direct)
95	DB_COPY2	Copy data area (indirect)
96	RETTEN	Save scratch flag (PLC 155U)
97	LADEN	Load scratchpad memory (S5-155U)
98	COD_B8	BCD-binary conversion 8 decades
99	COD_32	Binary-BCD conversion 8 decades

Analog functions

FC	Name	Designation
100	AE_460_1	Analog input module 460
101	AE_460_2	Analog input module 460
102	AE_463_1	Analog input module 463
103	AE_463_2	Analog input module 463
104	AE_464_1	Analog input module 464
105	AE_464_2	Analog input module 464
106	AE_466_1	Analog input module 466
107	AE_466_2	Analog input module 466
108	RLG_AA1	Analog output module
109	RLG_AA2	Analog output module
110	PER_ET1	ET100 distributed I/O
111	PER_ET2	ET100 distributed I/O

Math functions

FC	Name	Designation
112	SINUS	Sine
113	COSINUS	Cosine
114	TANGENS	Tangent
115	COTANG	Cotangent
116	ARCSIN	Arc sine
117	ARCCOS	Arc cosine
118	ARCTAN	Arc tangent
119	ARCCOT	Arc cotangent
120	LN_X	Natural logarithm
121	LG_X	Logarithm to base 10
122	B_LOG_X	Logarithm to any base
123	E_H_N	Exponential function with base e
124	ZEHN_H_N	Exponential function with base 10
125	A2_H_A1	Exponential function with any base

25.5 TI-S7 Converting Blocks

FB	Name	Designation
80	LEAD_LAG	Lead/lag algorithm
81	DCAT	Discrete control time interrupt
82	MCAT	Motor control time interrupt
83	IMC	Index matrix comparison
84	SMC	Matrix scanner
85	DRUM	Event maskable drum
86	PACK	Collect/distribute table data

FC	Name	Designation
80	TONR	Latching ON delay
81	IBLKMOV	Transfer data area indirectly
82	RSET	Reset process image bit by bit
83	SET	Set process image bit by bit
84	ATT	Enter value in table
85	FIFO	Output first value in table
86	TBL_FIND	Find value in table
87	LIFO	Output last value in table
88	TBL	Execute table operation
89	TBL_WRD	Copy value from the table
90	WSR	Save datum

91	WRD_TBL	Combine table element
92	SHRB	Shift bit in bit shift register
93	SEG	Bit pattern for 7-segment display
94	ATH	ASCII-hexadecimal conversion
95	HTA	Hexadecimal-ASCII conversion
96	ENCO	Least significant set bit
97	DECO	Set bit in word
98	BCDCPL	Generate ten's complement
99	BITSUM	Count set bits
100	RSETI	Reset PQ byte by byte
101	SETI	Set PQ byte by byte
102	DEV	Calculate standard deviation
103	CDT	Correlated data tables
104	TBL_TBL	Table combination
105	SCALE	Scale values
106	UNSCALE	Unscale values

25.6 PID Control Blocks

FB	Name	Designation
41	CONT_C	Continuous control
42	CONT_S	Step control
43	PULSGEN	Generate pulse
58	TCONT_CP	Continuous temperature control
59	TCONT_S	Step temperature control

25.7 Communication Blocks

FB	Name	Designation
8	USEND	Uncoordinated send
9	URCV	Uncoordinated receive
12	BSEND	Block-oriented send
13	BRCV	Block-oriented receive
14	GET	Read data from partner
15	PUT	Write data to partner
28	USEND_E	Uncoordinated send
29	URCV_E	Uncoordinated receive
34	GET_E	Read data from partner
35	PUT_E	Write data to partner

FC	Name	Designation
1	DP_SEND	Send data
2	DP_RECV	Receive data
3	DP_DIAG	Diagnostics
4	DP_CTRL	Control
62	C_CNTR	Scan connection status

For DP standard slaves and PROFINET IO devices

FB	Name	Designation
20	GETIO	Read inputs
21	SETIO	Set outputs
22	GETIO_PA	Read inputs consistently
23	SETIO_PA	Set outputs consistently

IE communications

FB	Name	Designation
63	TSEND	Send data
64	TRCV	Receive data
65	TCON	Establish connection
66	TDISCON	Cancel connection
67	TUSEND	Send data via UDP
68	TURCV	Receive data via UDP

UDT Name	Designation
65 TCON_PAR	Data structure for connection configuration
651 TCON_PAR	TCP_conn_active
652 TCON_PAR	TCP_conn_passive
653 TCON_PAR	ISOonTCP_conn_active
654 TCON_PAR	ISOonTCP_conn_passive
655 TCON_PAR	ISOonTCP_conn_CP_active
656 TCON_PAR	ISOonTCP_conn_CP_passive
657 TCPN_PAR	UDP_local_open
66 TADD_PAR	Address structure of the communication partner
661 TADD_PAR	UDP_rem_address and port

FB	Name	Designation
210	FW_TCP	TCP server for FETCH/WRITE
220	FW_IOT	ISO-on-TCP server for FETCH/WRITE

25.8 Miscellaneous Blocks

FC	Name	Designation
60	LOC_TIME	Read local time and summer ID
61	BT_LT	Convert module time into local time
62	LT_BT	Convert local time into module time
63	S_LTINT	Set time interrupt according to local time
FB	Name	Designation
60	SET_SW	Switch over summer/winter time
61	SET_SW_S	Switch over summer/winter time with time status
62	TIMESTMP	Send messages with time stamp
UDTName		Designation
60	WS_RULES	Rules for summer/winter time switching

25.9 SIMATIC_NET_CP

Library program CP 300

FB	Name	Designation
		FMS communication:
2	IDENT	Identify partner
3	READ	Read data from partner
4	REPORT	Transmit variables
5	STATUS	Request status information from partner
6	WRITE	Write data to partner
8	USEND	Uncoordinated send
9	URCV	Uncoordinated receive
12	BSEND	Block-oriented send
13	BRCV	Block-oriented receive
14	GET	Read data from partner
15	PUT	Write data to partner
40	FTP_CMD	FTP commands (replaces FC 40 to 44)
52	PNIO_REC	Transmit data record
54	PNIOALRM	Receive interrupt
55	IP_CONF	Transmit configuration
56	LOG_TRIG	Trigger ERPC communication

FC	Name	Designation
1	DP_SEND	Send data
2	DP_RECV	Receive data
3	DP_DIAG	Diagnostics
4	DP_CTRL	Control
5	AG_SEND	Send data (PROFIBUS FDL and Industrial Ethernet)
6	AG_RECV	Receive data (PROFIBUS FDL and Industrial Ethernet)
7	AG_LOCK	Disable data exchange (Industrial Ethernet)
8	AG_UNLOC	Enable data exchange (Industrial Ethernet)
10	AG_CNTRL	Diagnose and initialize connections
11	PNIO_SND	Data transfer on PROFINET
12	PNIO_RCV	Data receipt on PROFINET
40	FTP_CONN	Establish connection to server
41	FTP_STOR	Send data block to server
42	FTP_RETR	Send file to client
43	FTP_DELE	Delete file on server
44	FTP_QUIT	Cancel connection
62	C_CNTRL	Scan connection status

UDTName		Designation
1	-	FILE_DB_HEADER

Library program CP 400

FB	Name	Designation
		FMS communications:
2	IDENT	Identify partner
3	READ	Read data from partner
4	REPORT	Send variables
5	STATUS	Request status information from partner
6	WRITE	Write data to partner
40	FTP_CMD	FTP commands (replaces FC 40 to 44)
55	IP_CONF	Send connection configuration

FC	Name	Designation
5	AG_SEND	Send data (PROFIBUS FDL and Industrial Ethernet)
6	AG_RECV	Receive data (PROFIBUS FDL and Industrial Ethernet)
7	AG_LOCK	Disable data exchange (Industrial Ethernet)
8	AG_UNLOC	Enable data exchange (Industrial Ethernet)
10	AG_CNTRL	Diagnose and initialize connections
40	FTP_CONN	Establish connection to server
41	FTP_STOR	Transmit data block to server
42	FTP_RETR	Transmit file to client
43	FTP_DELE	Delete file on server
44	FTP_QUIT	Cancel connection
50	AG_LSEND	Send data to PROFIBUS CP
53	AG_SSEND	Send data to Ethernet CP
60	AG_LRECV	Receive data from PROFIBUS CP
63	AG_SRECV	Receive data from Ethernet CP
UDT Name	Designation	
1	-	FILE_DB_HEADER

25.10 Redundant IO MGP V31

Supporting of module redundancy

Library program *Red_IO*

FB	Name	Designation
450	RED_IN	Read redundant I/O signals
451	RED_OUT	Output redundant I/O signals
452	RED_DIAG	Diagnose redundant I/O
453	RED_STAT	Read status of redundant I/O

FC	Name	Designation
450	RED_INIT	Initialize I/O redundancy
451	RED_DEPA	Trigger depassivation

25.11 Redundant IO CGP V40

Supporting of individual module channel redundancy

Library program *Red_IO*

FB	Name	Designation
450	RED_IN	Read redundant I/O signals
451	RED_OUT	Output redundant I/O signals
452	RED_DIAG	Diagnose redundant I/O
453	RED_STAT	Read status of redundant I/O

FC	Name	Designation
450	RED_INIT	Initialize I/O redundancy
451	RED_DEPA	Trigger depassivation

25.12 Redundant IO CGP V51

Supporting of individual module channel redundancy

Library program *Red_IO*

FB	Name	Designation
450	RED_IN	Read redundant I/O signals
451	RED_OUT	Output redundant I/O signals
452	RED_DIAG	Diagnose redundant I/O
453	RED_STAT	Reads status of redundant I/O

FC	Name	Designation
450	RED_INIT	Initialize I/O redundancy
451	RED_DEPA	Trigger depassivation

26 Function Set LAD

26.1 Basic Functions

Memory functions

Singe coil	<div>Binary operand </div>
Midline output	<div>Binary operand </div>
Set coil	<div>Binary operand </div>
Reset coil	<div>Binary operand </div>
SR box	<div>Binary operand </div>
RS box	<div>Binary operand </div>
Positive edge in the power flow	<div>Edge memory bit </div>
Negative edge in the power flow	<div>Edge memory bit </div>
Positive edge of an operand	<div>Binary operand </div>
Negative edge of an operand	<div>Binary operand </div>

Binary checks and combinations

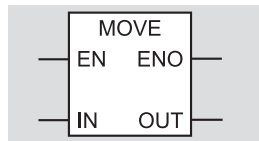
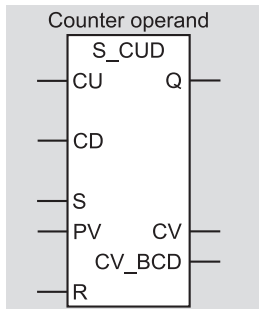
NO contact	<div>Binary operand </div>
NC contact	<div>Binary operand </div>
NOT contact	<div></div>

Timer functions

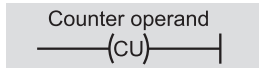
Timer box	<div>Timer operand </div>
Individual elements	
Start coil with time mode	<div>Timer operand </div> <div>Duration</div>
Reset coil	<div>Timer operand </div>
NO contact	<div>Timer operand </div>
NC contact	<div>Timer operand </div>

With the timer characteristics:

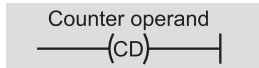
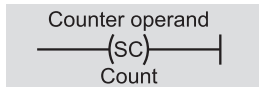
S_PULSE	SP	Pulse
S_PEXT	SE	Extended Pulse
S_ODT	SD	ON delay
S_ODTS	SS	Stored ON delay
S_OFFDT	SF	OFF delay

Transfer functions**MOVE box****Counter functions****Counter box****Individual elements**

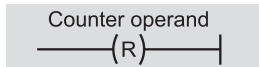
Count up coil



Count down coil

Set counter coil
with count

Reset coil



NO contact

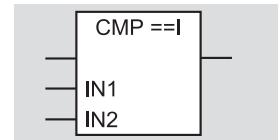


NC contact

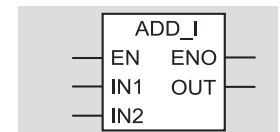


With the counter characteristics:

S_CUD Up/down counter
 S_CU Up counter
 S_CD Down counter

26.2 Digital Functions**Comparison functions****Comparison box**

Compare for	according to		
	INT	DINT	REAL
equal to	==I	==D	==R
not equal to	<>I	<>D	<>R
greater than	>I	>D	>R
greater than or equal to	>=I	>=D	>=R
less than	<I	<D	<R
less than or equal to	<=I	<=D	<=R

Arithmetic functions**Arithmetic box**

Calculation	according to		
	INT	DINT	REAL
Addition	ADD_I	ADD_DI	ADD_R
Subtraction	SUB_I	SUB_DI	SUB_R
Multiplication	MUL_I	MUL_DI	MUL_R
Division	DIV_I	DIV_DI	DIV_R
Modulo	-	MOD_DI	-

Mathematical Functions



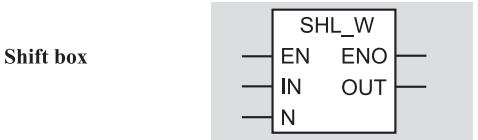
SIN	Sine
COS	Cosine
TAN	Tangent
ASIN	Arc sine
ACOS	Arc cosine
ATAN	Arc tangent
SQR	Finding the square
SQRT	Finding the square root
EXP	Establishing the exponent
LN	Finding the logarithm

Conversion Functions



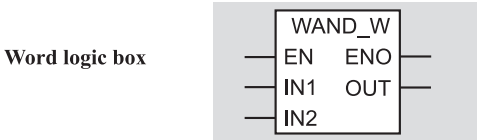
I_DI	Conversion of INT to DINT
I_BCD	Conversion of INT to BCD
DI_BCD	Conversion of DINT to BCD
DI_R	Conversion of DINT to REAL
BCD_I	Conversion of BCD to INT
BCD_DI	Conversion of BCD to DINT
CEIL	Conversion of REAL to DINT with rounding to next higher number
FLOOR	Conversion of REAL to DINT with rounding to next lower number
ROUND	Conversion of REAL to DINT with rounding to next whole number
TRUNC	Conversion of REAL to DINT without rounding
INV_I	INT one's complement
INV_DI	DINT one's complement
NEG_I	INT negation
NEG_DI	DINT negation
NEG_R	REAL negation
ABS	REAL absolute-value generation

Shift Functions



SHL_W	Shift word left
SHL_DW	Shift doubleword left
SHR_W	Shift word right
SHR_DW	Shift doubleword right
SHR_I	Shift word with sign
SHR_DI	Shift doubleword with sign
ROL_DW	Rotate left
ROR_DW	Rotate right

Word Logic



WAND_W	AND word
WOR_W	OR word
WXOR_W	Exclusive OR word
WAND_DW	AND doubleword
WOR_DW	OR doubleword
WXOR_DW	Exclusive OR doubleword

26.3 Program Flow Control

Status Bits

Result greater than zero	
Result greater than or equal to zero	
Result less than zero	
Result less than or equal to zero	
Result not equal to zero	
Result equal to zero	
Result invalid (unordered)	
Overflow	
Stored overflow	
Binary result	
SAVE coil	

Jump Functions

Jump if RLO = "1"	
Jump if RLO = "0"	
Entry, jump label	

Master Control Relay

Activate MCR area	
Deactivate MCR area	
Open MCR zone	
Close MCR zone	

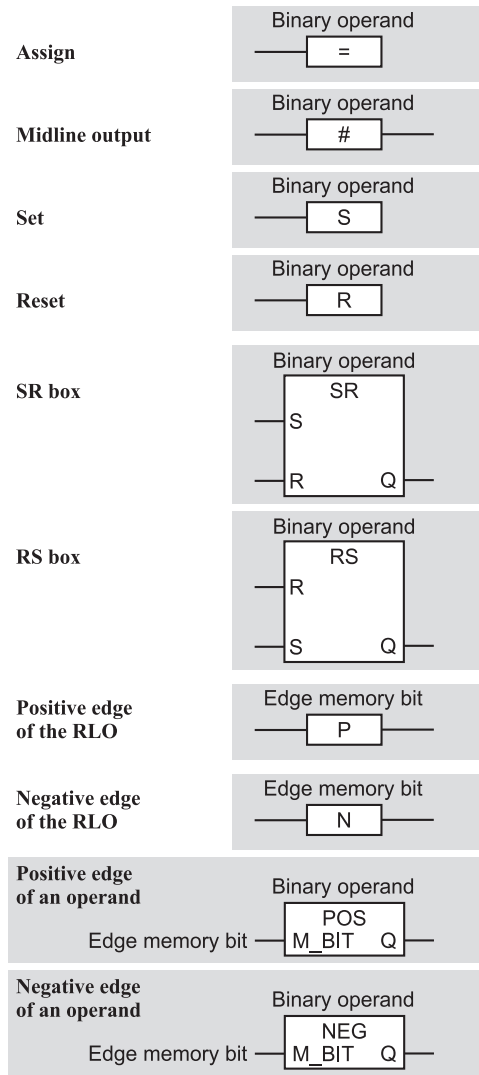
Block Functions

Calling a function block with data block	
Calling a system function block with data block	
Calling a function block or a system function block as local instance	
Calling a function	
Calling a system function	
Calling a non-parameterized function	
Calling a non-parameterized system function	
RET coil, conditional block end	
Open data block	

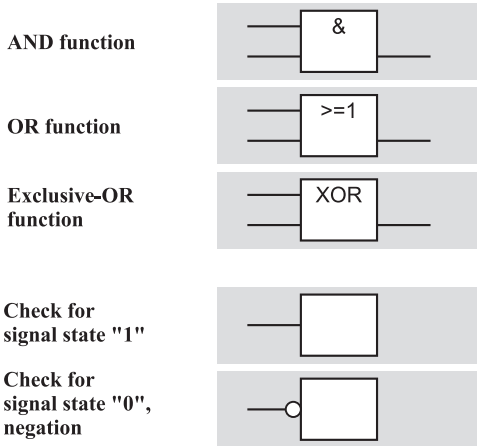
27 Function Set FBD

27.1 Basic Functions

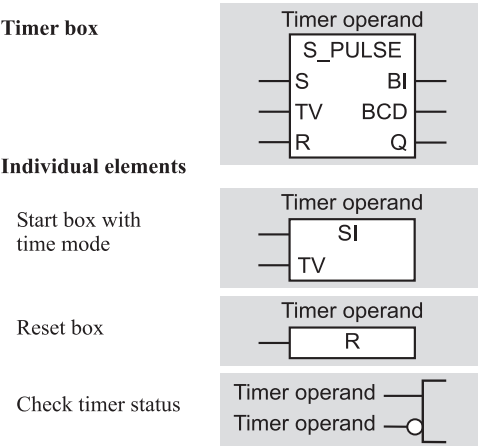
Memory Functions



Binary Checks and Combinations

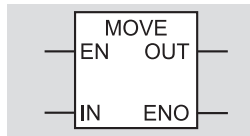
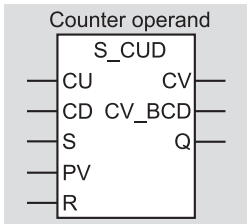


Timer Functions

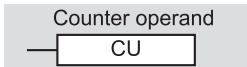


With the timer characteristics:

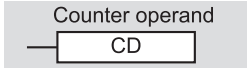
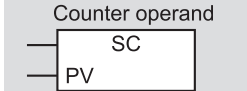
S_PULSE	SP	Pulse
S_PEXT	SE	Extended Pulse
S_ODT	SD	ON delay
S_ODTS	SS	Stored ON delay
S_OFFDT	SF	OFF delay

Transfer Functions**MOVE box****Counter Functions****Counter box****Individual elements**

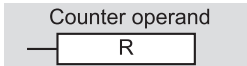
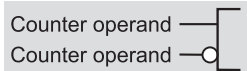
Count up box



Count down box

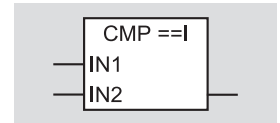
Set counter box
with count

Reset box

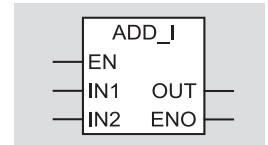
Check
counter status

With the counter characteristics:

S_CUD	Up/down counter
S_CU	Up counter
S_CD	Down counter

27.2 Digital Functions**Comparison Functions****Comparison box**

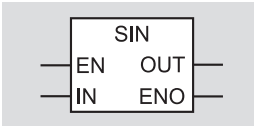
Compare for	according to		
	INT	DINT	REAL
equal to	==I	==D	==R
not equal to	<>I	<>D	<>R
greater than	>I	>D	>R
greater than or equal to	>=I	>=D	>=R
less than	<I	<D	<R
less than or equal to	<=I	<=D	<=R

Arithmetic functions**Arithmetic box**

Calculation	according to		
	INT	DINT	REAL
Addition	ADD_I	ADD_DI	ADD_R
Subtraction	SUB_I	SUB_DI	SUB_R
Multiplication	MUL_I	MUL_DI	MUL_R
Division	DIV_I	DIV_DI	DIV_R
Modulo	-	MOD_DI	-

Mathematical Functions

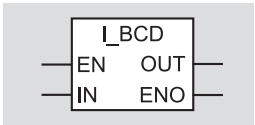
Math box



SIN	Sine
COS	Cosine
TAN	Tangent
ASIN	Arc sine
ACOS	Arc cosine
ATAN	Arc tangent
SQR	Finding the square
SQRT	Finding the square root
EXP	Establishing the exponent
LN	Finding the logarithm

Conversion Functions

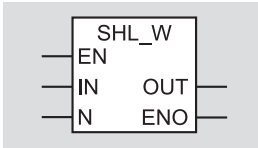
Conversion box



I_DI	Conversion of INT to DINT
I_BCD	Conversion of INT to BCD
DI_BCD	Conversion of DINT to BCD
DI_R	Conversion of DINT to REAL
BCD_I	Conversion of BCD to INT
BCD_DI	Conversion of BCD to DINT
	Conversion of REAL to DINT with rounding
CEIL	to next higher number
FLOOR	to next lower number
ROUND	to next whole number
TRUNC	without rounding
INV_I	INT one's complement
INV_DI	DINT one's complement
NEG_I	INT negation
NEG_DI	DINT negation
NEG_R	REAL negation
ABS	REAL absolute-value generation

Shift Functions

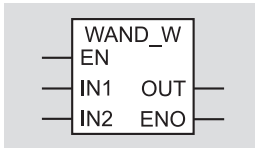
Shift box



SHL_W	Shift word left
SHL_DW	Shift doubleword left
SHR_W	Shift word right
SHR_DW	Shift doubleword right
SHR_I	Shift word with sign
SHR_DI	Shift doubleword with sign
ROL_DW	Rotate left
ROR_DW	Rotate right

Word Logic

Word logic box



WAND_W	AND word
WOR_W	OR word
WXOR_W	Exclusive OR word
WAND_DW	AND doubleword
WOR_DW	OR doubleword
WXOR_DW	Exclusive OR doubleword

27.3 Program Flow Control

Status Bits

Result greater than zero	<div>>0</div>
Result greater than or equal to zero	<div>>=0</div>
Result less than zero	<div><0</div>
Result less than or equal to zero	<div><=0</div>
Result not equal to zero	<div><>0</div>
Result equal to zero	<div>==0</div>
Result invalid (unordered)	<div>UO</div>
Overflow	<div>OV</div>
Stored overflow	<div>OS</div>
Check binary result	<div>BR</div>
Save binary result	<div>SAVE</div>

Jump Functions

Jump if RLO = "1"	<div>Destination JMP</div>
Jump if RLO = "0"	<div>Destination JMPN</div>
Entry, jump label	<div>Destination</div>

Master Control Relay

Activate MCR area	<div>MCRA</div>
Open MCR zone	<div>MCR<</div>
Close MCR zone	<div>MCR></div>
Deactivate MCR area	<div>MCRD</div>

Block Functions

Calling a function block with data block	<div>DBx FBx EN OUT1 IN1 OUT2 IN2 ENO</div>
Calling a system function block with data block	<div>DBx SFBx EN OUT1 IN1 OUT2 IN2 ENO</div>
Calling a function block or a system function block as local instance	<div>#name EN OUT1 IN1 OUT2 IN2 ENO</div>
Calling a function	<div>FCx EN OUT1 IN1 OUT2 IN2 ENO</div>
Calling a system function	<div>SFCx EN OUT1 IN1 OUT2 IN2 ENO</div>
Calling a non-parameterized function	<div>FCx CALL</div>
Calling a non-parameterized system function	<div>SFCx CALL</div>
Conditional block end	<div>RET</div>
Open data block	<div>DB x OPN</div>

Index

A

- Actual parameters 257
- Address priority 77
- Addressing
 - absolute 117
 - indirect 419
 - symbolic 118
- AND function 136
- ANY pointer
 - description 421
 - introduction 165
- Arc functions 204
- Archiving projects 56
- Arithmetic functions 197
 - status bits 222
- ARRAY (data type) 126
- Ascertaining a module address 394
- Assign
 - box (FBD) 146
- Asynchronous errors 408
- Automation License Manager 50

B

- Background scanning OB 90 275
- Binary flags 221
- Binary logic operations 131
 - FBD 134
- Binary result 222
 - EN/ENO 225
 - SAVE 225
 - status bit BR 222
- Bit memory 48
- Block functions for code blocks 235
- Block libraries 426
- Block parameters 252
 - forwarding 260
- Blocks
 - checking block consistency 104
 - comparing 77
 - correcting interface 103
 - description 98
 - opening 106
 - programming description 106

- programming FBD 111
- programming general 73
- programming LAD 110
- properties 100
- protection 103, 418
- structure 100
- transferring 84
- types 98

Box

- assign box 146
- memory box FBD 148
- memory box LAD 144
- reset box 148
- set box 148

C

- CHAR (data type) 122
- Checksum 58, 98
- Clock memory 48
- Coil
 - reset coil 142
 - set coil 142
 - single coil 142
- Cold restart 393
- Combining connections 70
- Comments multilingual 80
- Communication
 - distributed I/O 287
 - global data communication 331
 - IE communication 350
 - introduction 35
 - PtP communication 357
 - S7 basic communication 335
 - S7 communication 341
- Communication error OB 87 410
- Comparison functions 193
- Compressing
 - SFC 25 COMPRESS 282
 - user program 85
- Configuration in RUN 362
- Configuration table 60
- Configuring
 - connections 65

- Configuring stations 58
- Connecting a PLC 81
- Connection table 65
- Connections
 - loading data 69
- Controlling I/O bits 233
- Conversion functions 207
 - status bits 222
- Counter functions
 - IEC counters 186
 - SIMATIC counters 182
- CPU hardware faults OB 84 409
- CPU information 83
- Creating projects 54
- D**
 - Data block
 - functions 244
 - monitoring and modifying data
 - addresses 92
 - offline/online 85
 - open 246
 - programming 113
 - registers 244
 - windows and views 114
 - Data operands 245
 - Data types
 - complex 125
 - elementary 120
 - subdivision 119
 - user-defined 128
 - DATE (data type) 124
 - DATE_AND_TIME (data type) 126
 - Device name, device number 306
 - Diagnosing hardware 87
 - Diagnostic interrupt OB 82 412
 - Diagnostics address 44
 - PROFIBUS DP 289
 - PROFINET IO 307
 - Digital flags 221
 - DINT (data type) 122
 - DINT calculation 200
 - status bits 222
 - Direct data exchange *see* DP master system
 - Disable output modules 388
 - Distributed I/O
 - addressing PROFIBUS DP 287
 - addressing PROFINET IO 305
 - configuring PROFIBUS DP 291
 - configuring PROFINET IO 307
 - introduction PROFIBUS DP 29
 - introduction PROFINET IO 30
 - System blocks 323
- Down counting 185
- DP master system
 - configuring 292
 - description 29
 - direct data exchange 302
 - equidistant bus cycles 302
 - in network configuration 64
 - isochronous mode 302
- DPV1 interrupts 378
- E**
 - Edge evaluation
 - description 152
 - Editor
 - FBD elements 111
 - LAD elements 110
 - program 73
 - symbolic 71
 - EN/ENO mechanism 225
 - Enabling peripheral outputs 90
 - Error handling 404
 - Example
 - binary scaler 154
 - conveyor belt 260
 - conveyor control 156
 - counter control 188
 - editing the message frame 424
 - feed 262
 - indirect copying of a data area 424
 - message frame 422
 - parts counter 261
 - save message frame 425
 - time-of-day check 422
 - Exclusive OR function 138
 - Exponentiation 205
 - Extended pulse timer 176
- F**
 - Fault-tolerant SIMATIC 24
 - First check 221
 - Forcing variables 89
 - Formal parameters 255
 - Function set
 - FBD 440
 - LAD 436
- G**
 - Gateways 68
 - Global data communication 331
 - Global data table 333

GSD files

for PROFIBUS DP 300

for PROFINET IO 314

H

Hardware catalog 59

Hardware configuration 58

Hardware interrupts 376

I

IEC counter functions 186

IEC functions

library 430

IEC timer functions 180

Indirect addressing 419

Inputs 46

Insert/remove module interrupt OB 83 409

INT (data type) 122

INT calculation 199

status bits 222

Interrupt handling 367

DPV1 interrupts 378

general remarks 367

hardware interrupts 376

multiprocessor interrupt 380

synchronous cycle interrupts 381

time-delay interrupts 372

time-of-day interrupts 368

watchdog interrupts 374

IP address 305

Isochronous mode 302

J

Jump functions 227

L

Language setting 80

Libraries

communication blocks 433

creating 56

general 53

IEC function blocks 430

miscellaneous blocks 434

organization blocks 426

overview 426

PID control blocks 433

Redundant CGP V40 435

Redundant IO CGP V50 435

Redundant IO MGP V30 435

S5-S7 converting blocks 431

SIMATIC_NET_CP 434

system function blocks 427

TI-S7 converting blocks 432

Load memory 26

Local data

static 241

temporary 240

Local instances 242

Local time 278

Logarithm 205

Logical address 44

M

Main program OB1 270

Master Control Relay (MCR) 230

Mathematical functions 202

status bits 222

Memory bits 48

Memory box

FBD 148

LAD 144

Memory card 25

Memory functions

FBD 146

LAD 142

Memory reset 389

Micro memory card 26

Midline outputs 150

Minimum scan cycle time 275

Modifying variables 89

Module time 278

Modules

ascertaining addresses 394

monitoring and modifying 62

parameterizing 397

start address 44

Monitoring variables 89

MOVE box 162

Move functions 161

Multilingual texts 80

Multiple instance *see Local instances*

Multiprocessor

interrupt 380

operation 282

Multiproject

adjust projects 69

creating and editing 57

N

NC contact 131

Negation

conversion functions 212

normally closed (NC) contact 131

- NOT contact (LAD) 134
- RLO 139
- scanning for "0" (FBD) 135
- Network
 - configuration 62
 - templates 108
- NO contact 131
- Number range overflow 222
- O**
 - off-delay timer 179
 - on-delay timer 177
 - Online help 54
 - Open data block 246
 - Operating modes
 - HOLD 388
 - RESTART 391
 - RUN (main program) 270
 - START-UP 391
 - STOP 389
 - Operating modes of the CPU 387
 - OR function 137
 - Organization blocks
 - asynchronous errors OB 80 to OB 87 408
 - background scanning OB 90 275
 - determining runtime 283
 - diagnostic interrupt OB 82 412
 - interrupts OB 10 to OB 64 367
 - main program OB 1 270
 - overview 97
 - redundancy error OB 70 to 73 410
 - restart OB 100 to OB 102 388
 - synchronous errors OB 121, OB 122 404
 - Outputs 47
 - Overflow 222
- P**
 - Parallel circuits 132
 - Parameter types 128
 - Parameterizing modules 397
 - Peripheral inputs 46
 - Peripheral outputs 46
 - Pointer
 - general remarks 419
 - Power supply errors OB 81 409
 - Priority classes 96
 - Process image
 - description 46
 - isochrone updating 382
 - subprocess images 272
 - updating 272
 - Processing abort OB 88 410
 - PROFIBUS DP
 - addressing 287
 - configuring 291
 - isochronous mode 302
 - PROFINET IO
 - addressing 305
 - configuring 307
 - send cycle time/update time 315
 - sync domain 317
 - topology editor 317
 - PROFINET IO system
 - description 31
 - in network configuration 64
 - Program editor
 - FBD elements 111
 - general 73
 - LAD elements 110
 - Program elements catalog 107
 - Program execution errors OB 85 409
 - Program length 98
 - Program organization 271
 - Program processing methods 94
 - Program status 91
 - Program structure 270
 - Programming code blocks 106
 - Programming data blocks 244
 - Programming networks 108
 - Project
 - archiving 56
 - creating 54
 - general 53
 - object hierarchy 52
 - Project versions 57
 - Protecting
 - user program 286
 - PtP communication 357
 - Pulse timer 175
- R**
 - Rack failure OB 86 410
 - REAL (data type) 124
 - REAL calculation 200
 - status bits 222
 - Reference data 78
 - Reset box 148
 - Reset coil 142
 - Reset function 148
 - Response time 276
 - Restart characteristics 387
 - Restart types 391

Result of the logic operation 221
 negation (FBD) 139
 negation (LAD) 134
Retentive on-delay timer 178
Retentivity 390
Rewiring 77
Rotate functions 216
Rounding 210
RS memory function
 FBD 150
 LAD 144
Run-time meter 280

S

S5-S7 converting blocks
 libraries 431
S5TIME (data type) 124
S7 basic communication
 external 338
 internal 335
S7 communication 341
S7-300 station 20
S7-400 station 22
Safety-related SIMATIC 24
Scan cycle monitoring time 274
Scan cycle statistics 274
Sensor type 139
Series circuits 132
Set box 148
Set coil 142
SFB 0 CTU 187
SFB 1 CTD 187
SFB 104 IP_CONF 331
SFB 12 BSEND 344
SFB 13 BRCV 344
SFB 14 GET 344
SFB 15 PUT 344
SFB 16 PRINT 345
SFB 19 START 346
SFB 2 CTUD 187
SFB 20 STOP 346
SFB 22 STATUS 347
SFB 23 USTATUS 347
SFB 3 TP 180
SFB 4 TON 180
SFB 5 TOF 180
SFB 52 RDREC 401
SFB 53 WRREC 402
SFB 54 RALRM 385
SFB 60 SEND_PTP 358
SFB 61 RCV_PTP 359
SFB 62 RES_RCVB 359
SFB 63 SEND_RK 361
SFB 64 FETCH_RK 361
SFB 65 SERVE_RK 362
SFB 75 SALRM 326
SFB 8 USEND 343
SFB 81 RD_DPAR 399
SFB 9 URC 343
SFC 0 SET_CLK 278
SFC 1 READ_CLK 278
SFC 100 SET_CLKS 278
SFC 101 RTM 281
SFC 102 RD_DPARA 399
SFC 103 DP_TOPOL 329
SFC 104 CIR 365
SFC 109 PROTECT 286
SFC 11 DPSYN_FR 328
SFC 12 D_ACT_DP 325
SFC 13 DPMRM_DG 329
SFC 14 DPRD_DAT 325
SFC 15 DPWR_DAT 325
SFC 2 SET_RTM 281
SFC 20 BLKMOV 165
SFC 21 FILL 165
SFC 22 CREAT_DB 248
SFC 23 DEL_DB 248
SFC 24 TEST_DB 248
SFC 25 COMPRESS 282
SFC 26 UPDAT_PI 273
SFC 27 UPDAT_PO 273
SFC 28 SET_TINT 370
SFC 29 CAN_TINT 371
SFC 3 CTRL_RTM 281
SFC 30 ACT_TINT 371
SFC 31 QRY_TINT 371
SFC 32 SRT_DINT 373
SFC 33 CAN_DINT 374
SFC 34 QRY_DINT 374
SFC 35 MP_ALM 381
SFC 36 MSK_FLT 407
SFC 37 DMSK_FLT 408
SFC 38 READ_ERR 408
SFC 39 DIS_IRT 383
SFC 4 READ_RTM 281
SFC 40 EN_IRT 384
SFC 41 DIS_AIRT 384
SFC 42 EN_AIRT 385
SFC 43 RE_TRIGR 274
SFC 44 REPL_VAL 408
SFC 46 STP 282
SFC 47 WAIT 282

- SFC 48 SNC_RTCB 278
 - SFC 49 LGC_GADR 395
 - SFC 5 GADR_LGC 395
 - SFC 50 RD_LGADR 397
 - SFC 51 RDSYSST 414
 - SFC 52 WR_USMSG 411
 - SFC 54 RD_DPARM 400
 - SFC 55 WR_PARM 400
 - SFC 56 WR_DPARM 401
 - SFC 57 PARM_MOD 401
 - SFC 58 WR_REC 403
 - SFC 59 RD_REC 402
 - SFC 6 RD_SINFO 277
 - SFC 60 GD_SND 335
 - SFC 61 GD_RCV 335
 - SFC 62 CONTROL 347
 - SFC 64 TIME_TCK 280
 - SFC 65 X_SEND 340
 - SFC 66 X_RCV 340
 - SFC 67 X_GET 340
 - SFC 68 X_PUT 340
 - SFC 69 X_ABORT 341
 - SFC 7 DP_PRAL 327
 - SFC 70 LOG_GEO 395
 - SFC 71 GEO_LOG 395
 - SFC 71 GEO_LOG 395
 - SFC 72 I_GET 337
 - SFC 73 I_PUT 337
 - SFC 74 I_ABORT 337
 - SFC 78 OB_RT 283
 - SFC 79 SET 233
 - SFC 80 RSET 233
 - SFC 81 UBLKMOV 165
 - SFC 82 CREA_DBL 248
 - SFC 83 READ_DBL 165
 - SFC 84 WRIT_DBL 165
 - SFC 85 CREA_DB 248
 - SFC 87 C_DIAG 348
 - SFC 99 WWW 416
 - Shift functions 213
 - status bits 224
 - SIMATIC counter functions 182
 - SIMATIC manager 50
 - SIMATIC timer functions 170
 - Single coil 142
 - Slot address 43
 - Source files
 - updating or generating 76
 - Square-root extraction 204
 - Squaring 204
 - SR memory function
 - FBD 149
 - LAD 144
 - Start information
 - DPV1 interrupts 379
 - hardware interrupts 377
 - interrupt handling 368
 - main program OB 1 276
 - multiprocessor interrupt 380
 - restart 388
 - synchronous cycle interrupts 382
 - temporary local data 241
 - time-delay interrupts 372
 - time-of-day interrupts 369
 - watchdog interrupts 374
 - Static local data 241
 - Status bits
 - binary result BR 222
 - description 221
 - evaluating 224
 - first check 221
 - OR 222
 - overflow OV 222
 - RLO 221
 - setting 222
 - status 221
 - stored overflow OS 222
 - Stored overflow 222
 - STRING (data type) 126
 - STRUCT (data type) 128
 - Subnets 37
 - Subprocess images 272
 - Symbol table 71
 - SYNC/FREEZE 300
 - Synchronous cycle interrupts 381
 - Synchronous errors 404
 - System blocks
 - description 100
 - System clock 280
 - System diagnostics 411
 - System function blocks
 - libraries 427
 - System memory 27
- ## T
- Temporary local data 240
 - TIME (data type) 124
 - Time of day 278
 - Time stamp conflict 104
 - TIME_OF_DAY (data type) 125
 - Time-based reaction
 - off-delay timer SFB 180

- on-delay timer SFB 180
- pulse timer SFB 180
- Time-delay interrupts 372
- Time-of-day interrupts 368
- Timer characteristics
 - extended pulse timer 176
 - off-delay timer 179
 - on-delay timer 177
 - pulse timer 175
 - retentive on-delay timer 178
- Timer functions
 - IEC timers 180
 - SIMATIC timers 170
- Timing errors OB 80 409
- Trigonometric functions 204

U

- UDT (data type) 128
- Up counting 185
- User blocks 98

- User data area 45
- User data types 128
- User program
 - compressing 85
 - load 83
 - protection 82
 - testing 91

V

- Variable declaration table 106
- Variable table 87

W

- Warm restart 393, 394
- Watchdog interrupts 374
- Word logic
 - description 217
 - status bits 224
- Work memory 27

Abbreviations

AI	Analog input	LAD	Ladder diagram
AO	Analog output	MC	Memory card
AS	Automation system	MCR	Master control relay
AS-I	Actuator-sensor interface	MMC	Micro memory card
BR	Binary result	MPI	Multipoint interface
CFC	Continuous function chart	OB	Organization block
CP	Communications processor	OP	Operator panel
CPU	Central processing unit	PG	Programming device
DB	Data block	PLC	Programmable controller
DI	Digital input (module)	PS	Power supply
DO	Digital output (module)	RAM	Random access memory
DP	Distributed I/O	RLO	Result of logic operation
DR	Data record	SCL	Structured control language
EPROM	Erasable programmable read-only memory	SDB	System data block
FB	Function block	SFB	System function block
FBD	Function block diagram	SFC	System function call
FC	Function call	SM	Signal module
FEPROM	Flash erasable programmable read-only memory	SSL	System status list
FM	Function module	STL	Statement list
IM	Interface module	UDT	User data type
		VAT	Variable table